National Defence | Défense nationale

REAL-TIME INTERPROCESSOR SERIAL COMMUNICATIONS SOFTWARE FOR SKYNET EHF TRIALS

by

Robin Addison

19941212 060

DEFENCE RESEARCH ESTABLISHMENT OTTAWA

REPORT N0. 1227

Canada

July 1994
Ottawa

National Defence / Défense nationale

REAL-TIME INTERPROCESSOR SERIAL COMMUNICATIONS SOFTWARE FOR SKYNET EHF TRIALS

by

Robin Addison
MILSATCOM Group
Space Systems and Technology Section
Radar and Space Division

DEFENCE RESEARCH ESTABLISHMENT OTTAWA
REPORT N0. 1227

PCN
041LM

July 1994
Ottawa

Abstract

The Skynet EHF (extremely high frequency) Trials consisted of several week-long accesses over Skynet 4A during 1993. The whole link (from transmitting ground terminal to Skynet to receiving ground terminal) was used to simulate an EHF downlink from a payload to a ground terminal. Use of the Skynet satellite allowed the experimentation at EHF with the ground terminal and payload simulators over a link that had real satellite effects such as link degradations caused by satellite motion and weather. To conduct the trials, it was recognized that many tasks needed to be active at once: pointing of antennas, monitoring power levels, synchronization, data communications and result logging. To shorten development time and simplify integration requirements, a distributed multiple computer processing system was chosen.

This paper describes the communications software which provided the services necessary for the distributed processing used in the trials. The challenge was to develop a system that was easy to integrate with the user software as well as to ensure that the communications hardware and software did not conflict with special purpose boards in the various computers. For simplicity, stop-and-wait ARQ (automatic repeat request) protocol was used for high-level message passing. Low-level communications services that do not require handshaking, were also provided for equipment control. The communications software package met these challenges and after extensive testing, was proven to provide the necessary communications among all the processors and special devices of the distributed system.

Résumé

Les essais Skynet en EHF (extrêmement haute fréquence) consistant en plusieurs périodes d'utilisation d'une durée d'une semaine chacune, ont eu lieu en 1993. Un lien unidirectionnel satellite-terre a été simulé par un lien composé d'une station terrestre émettrice, remplaçant la charge utile, d'un satellite, et d'un station terrestre réceptrice. L'utilisation du satellite Skynet a permis à CRDO (Centre de recherche pour la défense, Ottawa) de faire des expériences sur certains problèmes de communications par satellite comme les dégradations causées par le mouvement du satellite et les conditions météorologiques. Pour les essais, il a été nécessaire de faire plusieurs tâches en même temps: modification des azimuts des antennes, mesurage des niveaux des signaux, synchronisation en espace, temps et fréquence, communication des donnés, et enregistrement des résultats. Un système de traitement distribué a été choisi pour minimiser le temps de développement nécessaire.

Ce rapport décrit le logiciel pour les communications entre les ordinateurs durant les essais Skynet en EHF. Le défi était de développer un système de communications qui serait facile à intégrer avec les logiciels résidents et les cartes installées dans les ordinateurs. Le protocole "stop-and wait ARQ" a été choisi pour les communications de haut niveau entre les processeurs. Chaque message doit être reçu et sa réception accusée avant la transmission du prochain. Les services de communications de bas niveau ont été fournis pour le contrôle des instruments. Le logiciel présenté dans cet ouvrage a atteint son but en fournissant les communications entre les ordinateurs et entre les différents instruments utilisés pour les essais Skynet en EHF.

Executive Summary

The Skynet EHF (extremely high frequency) Trials consisted of several week-long accesses over Skynet 4A during 1993. The whole link (from transmitting ground terminal to Skynet to receiving ground terminal) was used to simulate an EHF downlink from a payload to a ground terminal. Thus, the transmitter was acting as the payload and the receiver was acting as the ground terminal. Use of the Skynet satellite allowed the experimentation at EHF with the ground terminal and payload simulators over a link that had real satellite effects such as link degradations caused by satellite motion and weather.

To conduct these trials, it was recognized that many tasks needed to be active at once: pointing of antennas, monitoring power levels, synchronization, data communications and result logging. To shorten development time, rather than integrating these tasks into one big multi-tasking computer, a distributed processing system was chosen. This allowed each of the processes to be developed independently and ensured that the many specialized hardware boards would not conflict with one-another. Though the tasks were split into multiple platforms, it was still necessary for them to be able to intercommunicate.

Asynchronous communications software is described which provided the services necessary for the distributed processing used in the trials. The challenge was to develop a system that was easy to integrate with the user software and to ensure that the communications hardware and software did not conflict with special purpose boards in the various computers. Two types of services are provided: high-level communications involving robust message handling with error free transmissions and low-level communications for controlling equipment.

For simplicity, stop-and-wait ARQ (automatic repeat request) protocol is used for high-level message passing. Each message must be received properly and acknowledged prior to the next message. Lost or corrupted messages are retransmitted until received without errors. To simplify debugging, but at the expense of efficiency, only printable characters are used for the messages and framing.

Because the communications software took control of all serial ports, low-level communications services which do not require handshaking were provided for equipment control. This facilitated the development of user software to command equipment such as antenna controllers through a serial port.

The software was developed using Microsoft C 6.0 on a Dell 433E running DOS 5.0 (Disk Operating System version 5.0) and the real-time hardware interface portion was written in assembly language. The communications software runs on any PC (personal computer) compatible computer though AT-class machines cannot operate their serial ports at the highest speeds.

The communications software met the challenge and, after extensive testing, was proven to provide the necessary communications among all the processors and special devices of the distributed system.

Table of Contents

Notational Conventions

The following notational conventions are used to aid in the specification of syntax as distinct from the normal text:

COM.C	Filename
TO=COM1	Literal - type exactly as shown
open_com	Software routine
number_errors	Item to be filled in/replaced with a value
{A | B}	Choose one (and only one) of the members of this group
CR	Control characters (CR = carriage return, LF = linefeed)
△	Literal space
`int c = 0;`	Software listings

Acknowledgments

I would like to thank the people at Defence Research Agency in the United Kingdom for their support and the use of the Skynet 4A satellite. Without the use of the EHF facility on the satellite, arranged through TTCP STP-6 (The Technical Cooperation Program, Technical Panel S6) working group, this project would never have been realized.

1. Introduction

1.1 Background

The MILSATCOM (military satellite communications) group at DREO (Defence Research Establishment Ottawa) and the Satellite Applications and Projects Directorate at CRC (Communications Research Centre) have been engaged in the study of EHF (extremely high frequency) frequency-hopped satellite communications for several years. Both groups provide support to the EHF SATCOM Project, a 48 million dollar project. Approximately 80% of this project is devoted to an EHF system simulator designated FASSET (functional advanced development model of a satellite system for evaluation and test) developed in industry. To analyze aspects of frequency hopping communications and synchronization, other than those used in FASSET, payload and ground terminal simulators have been developed in-house.

It became known, through participation in TTCP STP-6 (The Technical Cooperation Program, Technical Panel S6) workshops, that the EHF portion of Skynet 4A was available to other TTCP participants for experiments. Upon acceptance of the Canadian proposal for the Skynet EHF Trials by the British, the ground terminal and payload simulators were modified to allow the Skynet 4A satellite to be used as an EHF to X-band bent-pipe repeater. This allowed the experimentation at EHF with the simulators over a link that had real satellite effects such as link degradations caused by satellite motion and weather.

1.2 Skynet EHF Trials

The Skynet EHF Trials consisted of several week-long accesses over Skynet 4A during 1993. The transmitter was situated at CRC and the receiver at DREO. The whole link (from CRC to Skynet to DREO) was used to simulate an EHF downlink from a payload to a ground terminal. Thus, the transmitter at CRC was acting as the payload and the receiver at DREO was acting as the ground terminal. Skynet was used to introduce real satellite effects (such as doppler) to the link.

From the beginning, it was recognized that many tasks needed to be active at once: pointing of antennas, monitoring power levels, synchronization, data communications and result logging. To shorten development time, rather than integrating all these tasks into one big multi-tasking computer, a distributed processing system was chosen. This allowed each of the processes to be developed independently - often by different people. It also ensured that the many specialized hardware boards would not conflict with one-another as they could be put in different computers. Though the tasks were split into multiple platforms, it was still necessary for them to be able to communicate. Using existing ground terminal equipment, it was not possible to co-locate the transmitter and receiver. This separation of 1 km between the two further complicated the inter-processor communications.

1.2.1 Skynet EHF Trials Block Diagram

Fig. 1. shows the Skynet EHF trials block diagram. Normal rectangles represent off-the-shelf equipment and custom circuitry whereas the rounded rectangles indicate computers and processors hosts. Between boxes are three types of lines indicating the flow of information: data/control flow is represented by thin lines with small arrowheads, analog/RF (radio frequency) connections are represented by thick lines with hollow arrowheads and asynchronous serial communications are represented by the dashed lines with solid arrowheads. It is these asynchronous serial communication links that are provided

by the software documented herein.

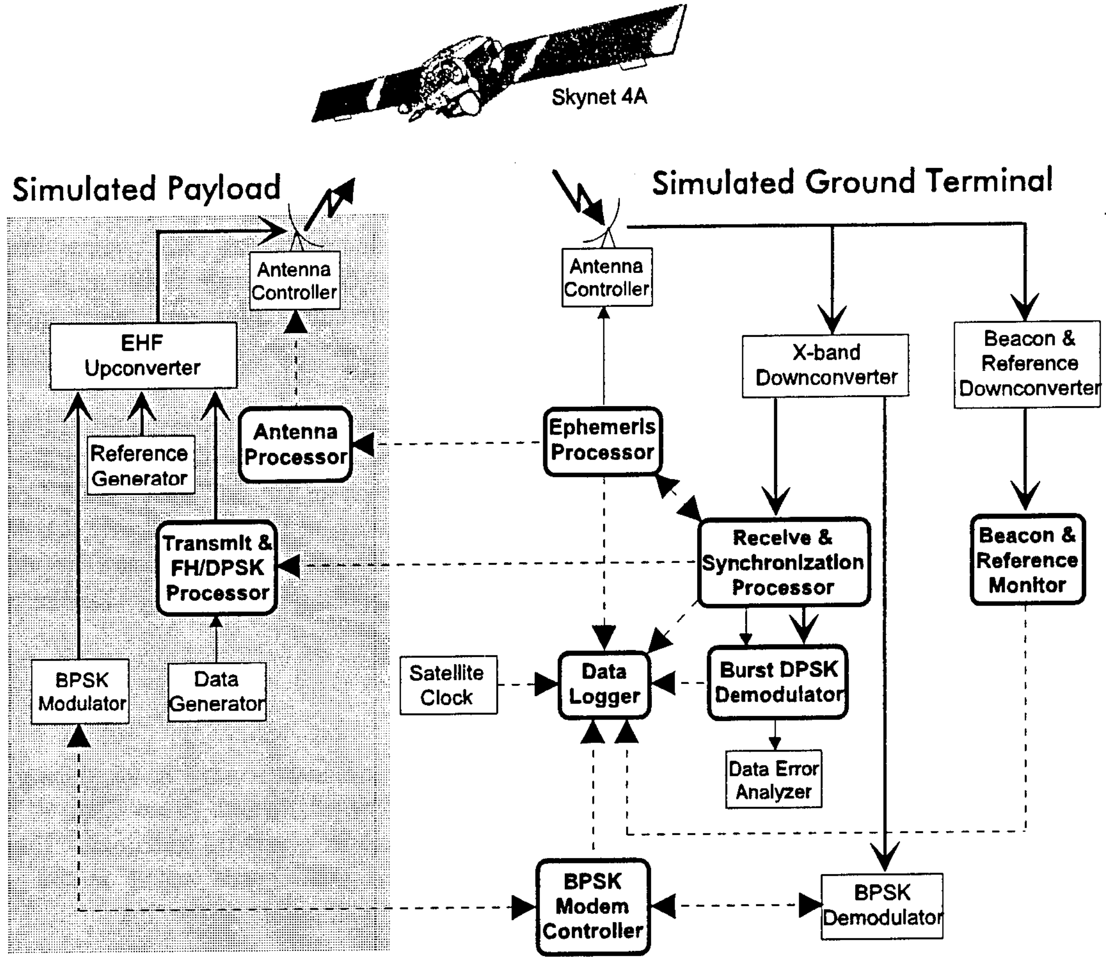

Fig. 1. Skynet EHF trials block diagram.

1.2.2 Normal Signal Flow

The primary signal flow starts at the ground terminal that is acting as the payload. Pseudo-random data from the Data Generator is passed to the Transmit & FH/DPSK Processor (FH/DPSK is frequency-hopped differential phase-shift keying) which performs data modulation and provides the frequency hopped pattern to the EHF Upconverter. Here, the hopping signal is combined with a reference signal provided by the Reference Generator (this signal is monitored at the receiver and is used to separate real uplink effects from that of the real downlink). This composite signal is then transmitted at EHF to Skynet 4A. On-board the satellite, the signal is translated and retransmitted at X-band.

The other ground terminal (which is acting as the ground terminal for the experimental link) receives the X-band signal and then processes it through the X-band Downconverter. The resultant downconverted signal is fed to the Receive & Synchronization Processor for synchronization processing and the signal is also passed on, with clocking, to the Burst DPSK Demodulator. The demodulated data is then fed into to Data Error Analyzer for bit-error-rate (BER) measurements. In the case of digital

voice, the Data Generator and the Data Error Analyzer were replaced with vocoders. The X-band downlink also contains the translated reference signal and a satellite beacon which are downconverted by the Beacon & Reference Downconverter and then measured by the Beacon & Reference Monitor.

1.2.3 Channel-characterization Signal Flow

To characterize the channel, unhopped BPSK (binary phase-shift keying) was used. This was done on the transmit side by replacing the hopped signal with an unhopped BSPK signal from a commercial satellite modem. After downconversion on the receive side, the signal is split off and fed to a similar unit for demodulation. These modems have built-in BER measurement capability. The modems are configured and monitored by the BPSK Modem Controller.

For antenna pointing information, the ephemeris information is generated by the Ephemeris Processor. For antenna scans, the pointing information is passed to the Receive & Synchronization Processor, modified with scan information, and then returned to the Ephemeris Processor. Antenna pointing is done by the receive Antenna Controller which is commanded by the Ephemeris Processor. The Ephemeris Processor also remotely commands the Antenna Processor on the transmit side, which in turn commands the transmit Antenna Controller.

1.2.4 Data Logging

Central to the whole system is the Data Logger. This computer logs data and status from five processors. It also gets the time from the GOES (Geostationary Operational Environmental Satellite) Satellite Synchronized Clock. Measurement data is sent from the Beacon & Reference Monitor several times each minute. The Ephemeris Processor routinely sends the pointing and predicted doppler values to the Data Logger. The Receive & Synchronization Processor sends raw synchronization data as well as synchronization performance measurements. Both the BPSK Modem Controller and the Burst DPSK Demodulation send BER measurements to the Data Logger.

1.2.5 Serial Communications

There are two types of asynchronous serial communications used for the experiment. Low-level asynchronous serial communications, involving simple character/string reads and writes to devices, are used in two cases. Low-level communications are used by the Transmit Antenna Processor to control the Antenna Controller and by the Data Logger to get the time from the GOES Satellite Clock. All other serial communications (shown by dashed lines) in the block diagram are high-level communications using automatic-repeat-request (ARQ) error control. High-level communications only occur among computers/processors.

1.3 Outline

This report first examines the trade-offs and design of the protocol for high-level communications involving robust message passing. The next chapter deals with the design and implementation of the software. The last chapter of this report covers the testing and problems that were uncovered during its use.

A substantial portion of this report is contained in various appendices. Appendix A contains the user's guide to the communications software, both high and low-level. It includes a program example

that exploits several features of the communications software. Appendix B contains the programmer's reference for the communications software. These two appendices together provide all the necessary information for a programmer to use the communications software.

The real-time assembly routines, which control the various aspects of the hardware, are documented in Appendix C. These routines can be used separately to allow interrupt driven communications callable from C language. Finally Appendix D and E contain the software listings for the communications software and real-time routines respectively.

2. Protocol Design

2.1 Introduction

The implementation of the communications software depended on several factors: availability of commercial software, ease of programming, ease of debugging, performance of links, topology of the links and, most importantly, requirements of the experiment. In the following sections, these aspects will be examined in detail and the final selection will be outlined. The theory portion of this section draws heavily on [1].

2.2 Commercial Software vs In-house Development

There are several communications packages for inter-computer communications available on the market. The advantages and disadvantages of using a commercial package or developing in-house software are presented in the table below:

Development Method	Advantages	Disadvantages
Commercial Package	- Very little or no development	- Uncustomizable - Cannot be debugged/altered - May not work with other realtime tasks - Must be selected with care to ensure necessary features are available - May require special (and expensive) hardware
In-house Development	- Can be customized - Can be debugged/altered - programmer is available to integrate it with other tasks	- Long development time - Complexity of development is proportional to sophistication of the network

Since the software was to be integrated with other real-time software (such as analog-to-digital board drivers, digital signal processor interfaces and instrument bus controller drivers) it was decided to use in-house development. The availability of the source code and the ability to modify the interface and, in some cases, to accommodate unusual or undocumented features of other real-time driver software were the key deciding factors.

2.3 Network

The topology and interconnect method among the computers has a major effect on the development time and complexity. The methods considered were a local network (for example using ethernet), a star topology where all stations are connected to one hub that passes messages between stations and a point-to-point network where there is a dedicated link for every communication between computers.

Some of the various options using the easiest available medium are presented in the table below along with their advantages and disadvantages.

Topology	Medium	Advantages	Disadvantages
Local network (bus or ring)	Ethernet (or others)	- high speed and throughput - easy to add or remove stations	- excessive complexity for in-house development
Star	Serial	- minimize the number of links required - speed is a function of the serial link	- hub station has to handle all traffic - requires a hub (ie: an extra computer) - serial can be slow
Point-to-point interconnect	Serial	- no routing required by any station - easy to add or remove stations/links - speed is a function of the serial link	- many links are required - serial can be slow

Since simplicity and flexibility were more important than performance, the point-to-point interconnect topology was selected using the standard serial ports available on personal computers.

2.4 Protocol Definition

A commercial software package would include a defined protocol for communications. Since the communications software was to be developed in-house, an appropriate protocol had to be selected. The key points considered are detailed below.

2.4.1 Error Control

Some method is required to correct errors or to allow retransmission of data in the event that an error occurs. Forward error correction (FEC) codes introduce redundancy in the data to allow the receiver to correct errors. This technique requires an encoder and decoder - relatively complex to implement. Another technique is to use error detection coupled with automatic-repeat-request (ARQ). This scheme uses a check value appended to the transmitted message. This check is verified at the receiver and if the verification fails, errors are detected and retransmission of the erroneous message is requested. The latter scheme, using a checksum, was chosen because of ease of implementation.

2.4.2 Flow Control

To ensure that the receiver does not lose any data when the transmitter is sending data quickly, flow control is required. This can be accomplished by several methods including:

- Polling: The transmitter polls the receiver to see if it is ready
- Ready: The receiver indicates that it is ready for data
- Interrupt: The receiver interrupts the transmitter when there is too much data

Stop-and-wait includes a form of the Ready flow control because the receiver, upon receipt of a message, does not acknowledge it until ready for the next message. Stop-and-wait flow control was chosen because it is well integrated with the ARQ scheme for error control.

2.4.3 Control/Data Discrimination

In any protocol, it is necessary to distinguish between control messages (such as Ack, Nak and routing) and user data messages. This can be done by keeping all control information in headers, by

using special codes to indicate control messages or by using a different medium. For the serial communication system, it was decided that all user data messages will be prefixed with a header (which includes some control information) and that strictly control messages would not have this header. To distinguish between control and user data messages, the header will use characters that cannot occur in the control messages.

2.4.4 Character vs Bit-oriented Protocol

Bit-oriented protocols are more efficient than character-oriented protocols because only the number of bits needed are used whereas character-oriented protocols must use an integral number of bytes as the minimum allocation. When using asynchronous character-oriented serial ports, however, it is much simpler to use a character-oriented protocol. Because simplicity was more important than efficiency, a character-oriented protocol was selected. To simplify debugging, this protocol was further restricted to using only printable characters.

2.4.5 Synchronous vs Asynchronous

Synchronous serial communications is more efficient than asynchronous serial communications because of the capacity needed for start and stop bits in asynchronous communications. The disadvantage of synchronous serial communications is that a clock signal is required along with the data to clock the data bits. Asynchronous serial communications was chosen because it is simpler to wire and is commonly used on personal computers.

2.4.6 Frame Synchronization

It is important for the receiver to recognize the beginning and end of a message frame. The delimiter of the header indicates the start of the message (though this same character could be included in the data portion). To delimit the end of a message frame, carriage return/linefeed was used. These control characters cannot occur in the data portion so they provided an unambiguous indication of the end of the frame. The end of one frame also marks the beginning of the next because asynchronous communication does not have idle characters between messages.

2.4.7 Addressing

Given point-to-point topology wherever communications are required, there is no need for addressing of the messages (since any message received on a specific link can only come from the station at the other end of the link). It is possible that, in a future system, the complexity of a full point-to-point connection may prove to be impractical. In that case, it would be desirable to have addressing information to allow messages can be passed on by intermediate stations. To allow for expansion, addressing information was included in the message header.

2.5 Stop-and-wait ARQ

One method of error control on a communication link is ARQ. In this scheme, the transmitter sends a message with some form of checksum which is received and then verified. If the verification is successful, the message is acknowledged. If the verification fails, the receiver requests retransmission of the message. Common ARQ schemes are: selective repeat, go-back-N and stop-and-wait. Selective repeat, the most efficient, allows the transmitter to continually transmit messages without pausing for

acknowledgments and only the messages in error are retransmitted. In go-back-N, the transmitter continually transmits, but if an error occurs in a message, the transmitter must go back to that message and retransmit it and all succeeding messages. The simplest and least efficient form of ARQ is stop-and-wait ARQ where the transmitter sends only one message at a time and must wait for acknowledgement prior to transmitting the next message. Stop-and-wait ARQ was chosen for high-level communications.

2.5.1 Normal Messages

Fig. 2. shows the information flow for normal message transmissions and the cases where a single error occurs. The normal message case shows the transmitting station (Tx) sending message #0 (Msg0) to the receiving station (Rx). It takes a certain time to send the message, Rx processes the message checking for errors and then responds with the appropriate acknowledgement for message #0 (Ack0). Some time later, Tx has another message, message #1, and the same sequence occurs.

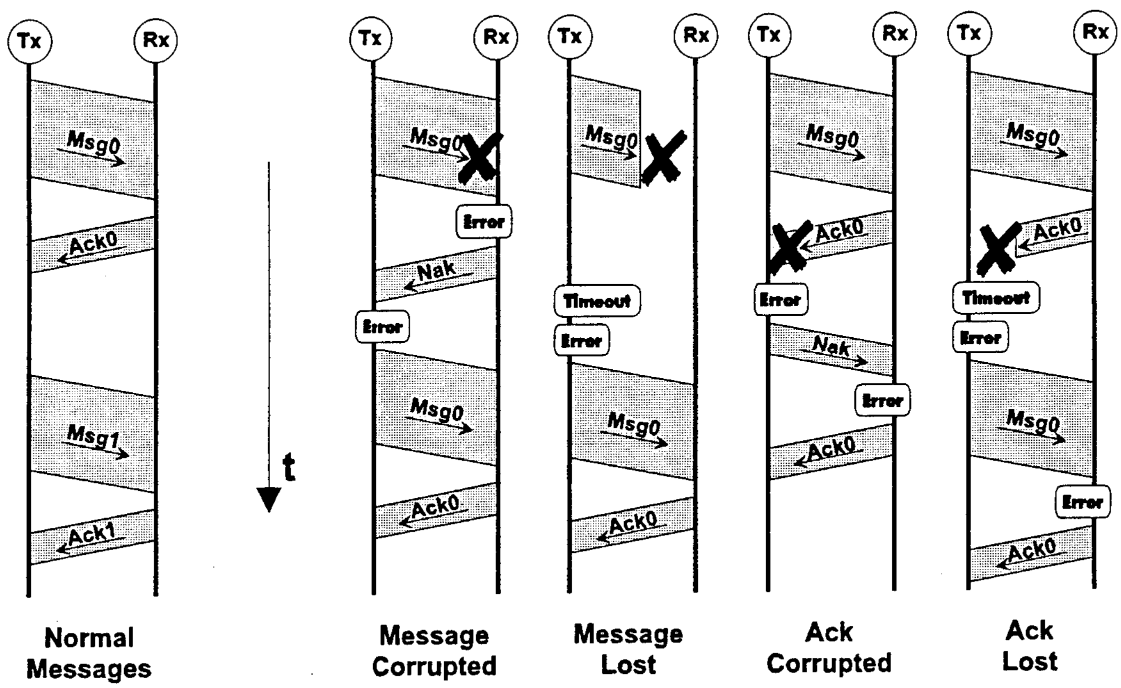

Fig. 2. Normal and single-error cases for stop-and-wait ARQ.

It is necessary that the acknowledgement number (but not negative acknowledgments) be matched up to the message number to distinguish between duplicate messages and lost messages. For stop-and-wait ARQ, it is only necessary to have two numbers to resolve the ambiguity - in the case of the diagram they are 0 and 1.

2.5.2 Single Errors

There are two cases of single-error events. A transmission could be corrupted (in which case the receiver gets some data, but with invalid framing or erroneous checksum) or a transmission could be missed completely. When the transmitted message is corrupted, the receiver first detects and reports a corrupted message. The receiver then responds with a negative acknowledgement (Nak). Upon receipt of the Nak, Tx reports an error condition (now both Tx and Rx have reported the corruption) and

retransmits the message. When the valid message is received, the appropriate Ack is generated by Rx. Once the Ack is received by Tx, the message has been passed error free and the protocol is complete.

When the entire message is lost, Rx sees no data at all and therefore, there is no Ack (nor a Nak) sent by Rx. Tx after having sent a message only waits for a limited time for the acknowledgement and after this period times-out, reports a message lost and retransmits the message. Rx responds with Ack and the message has been passed error free.

If the Ack is corrupted, Tx reports the error, responds with a Nak and then Rx reports an error and retransmits the Ack resulting the message being passed error free. If the Ack is lost completely, Tx times-out, reports the error and retransmits the message. Rx then receives a duplicate of a valid message so reports this error, acknowledges and then discards the duplicate message. Once again the message has been passed error free and without duplication.

2.5.3 Other Problems

2.5.3.1 Loss of Message Number Synchronization

Another event that could occur is the loss of synchronization between message number and acknowledgement number. In the case that the message or ack received is not the one expected, the receiver reports the error and switches the expected number to be in synchronization with the received message number. This event occurred often in the trials when the software on one machine was reset without resetting the connected machines. After one error report, the machines are back in synchronization.

2.5.3.2 Message or Ack Ambiguity

Another problem could occur when both stations are transmitting a message to each other at the same time. One station transmits a long message so the message is still being sent after the incoming short message has been received. After the long message has been sent, an acknowledgment to the received short message is transmitted. If the other station then sends a Nak (because of an error), there exists an ambiguity. The error could be caused by a corrupted long message or by a corrupted Ack for the short message. Since the long message originator cannot determine which caused the error, both the Ack and the long message are retransmitted. This will result in either a duplicate message error or and extra Ack error, but both the long and short messages will have been passed error free.

2.5.3.3 Multiple Errors

All other events require at least two errors to occur, and even in the case of multiple errors, the stations will remain synchronized. It is possible, with multiple errors, to lose a message without having detected the loss. But given the robustness of the physical link, such a sequence of errors are most improbable.

2.6 Implementation

Given that stop-and-wait ARQ is used for the protocol, the implementation details must be determined. In this section, first the factors affecting the implementation will be detailed, followed by the details of the format of messages.

2.6.1 **Factors Affecting Implementation**

2.6.1.1 Minimum Content of Message

Stop-and-wait protocol requires a message number (0 or 1) to distinguish between duplicate messages or loss of synchronization and also requires a checksum for error detection. User message data is an essential part of the message.

2.6.1.2 Message Numbering

To resolve ambiguities, two message numbers (0 and 1) are needed for stop-and-wait ARQ. Rather than including a message number field in the message headers and acknowledgements, the message numbering was implemented using the case (lower or upper) of key letter(s) to designate message number 0 or 1. For the message header, the case of the 'h' used in the checksum was set. For the acknowledgement, the case of the three letters were set. It is recognized that this implementation is a little cryptic, but it allowed for easy parsing of received messages and acknowledgments. A better implementation would have been to include a message number field in the header and acknowledgements.

2.6.1.3 Desirable Fields

For future expandability, possibly involving routing in a complex network, it is desirable to have the source and destination station names in the message header. It would be desirable to have a message type field to streamline the processing of messages.

2.6.1.4 Debugging Aids

This communications system was needed to support the Skynet EHF Trials - it was not an end to itself. Thus, it was desirable to minimize the development time, possibly at the expense of efficiency. To simplify debugging, the following features were selected:

- <u>Printable character messages ending with carriage return and linefeed</u>

  This choice ensures that a dumb terminal and a protocol analyzer could be used to debug the protocol. The negative aspects are that using only printable characters is inefficient for throughput (not a problem in this application) and that there are restrictions on the characters which can be included in the message.

- <u>Allow the checksum to be omitted</u>

  The receiver will not validate the checksum if it is "XX" instead of a hexadecimal number. During debugging, when it is desirable to generate a message by hand, one does not have to compute the checksum (a tedious and error prone task).

2.6.1.5 Fixed or Variable Length Fields

To simplify parsing, fixed length fields are desirable. This is true for the message text field, but such a restriction might impose undue constraints on the variety of messages, so a compromise was chosen. This compromise was to have fixed length header and a variable length text field.

2.6.2 Control Messages

The only valid control messages are listed below. ACK and ack acknowledge the receipt of a message with no errors and the case of the ACK/ack matches the case of the 'h' on the checksum of the transmitted message. Nak is used to request the retransmission of the message because of errors.

ack CR LF
ACK CR LF
nak CR LF

where CR LF is a carriage return and a linefeed to terminate the message

2.6.3 User Message Format

To pass data between machines, the user message is used. The two forms of the user message are given below (one with user message data and one with a null message):

[from_station **>** to_station **;** message_type **;** checksum **]** CR LF
[from_station **>** to_station **;** message_type **;** checksum **]** △ message_data CR LF

where:

[ ]	delimit the header
> ;	separators within the header
△	space character " " is only included when there is message data
CR LF	carriage return and linefeed to terminate the message
from_station	station field identifying the source of the message (see the table on the next page for valid station names); this field is 4 characters long
to_station	station field identifying the destination of the message (see the table on the next page for valid station names); this field is 4 characters long
message_type	message type field (see table below for valid message types); this field is 6 characters long and is blank filled if the message type is less than six characters
checksum	three character field comprised of two characters of hexadecimal checksum then an 'h' or 'H' (the case of the 'h' indicates whether "ack" or "ACK" is required)
message_data	optional variable-length message data, up to 199 characters plus the null terminator. If there is no data, then the preceding space is omitted. Message data should not include any control characters, especially not the carriage return and linefeed used to terminate a message.

Examples (checksums are only for illustrative purposes, they have not been calculated):

```
[sync>dlog;log   ;4Dh] Spatial scan complete at 10:51
[ephm>crca;point ;A2H] 10:58 12 Mar 93, Az=122.45, El=12.60, R=36132.8
[txpr>sync;status;22h]
```

Station Field		Message Type Field	
Value	Description	Value	Description
dlog	Data Logger & Experiment Controller	comd	Command message
beac	Beacon & Reference Monitor	config	Configuration message
bdem	Burst DPSK Demodulator Host	log	Log message
txpr	CRC Transmit Processor	status	Status message
ephm	Ephemeris Processor	point	Initial antenna pointing information
sync	Synchronization Processor	modpnt	Modified antenna pointing information
crca	CRC Antenna Controller Host	time	Time of day message
t85a	T85 Antenna Controller Host	error	Error condition message

2.6.4 Hardware Considerations

The communication system was implemented on the asynchronous serial ports of a PC (personal computer). Most computers involved only required one or two serial ports to be fully connected, but several computers needed more ports, one as high as eight ports. Ports beyond three were supplied using the Digiboard DigiCHANNEL PC/8 eight-port serial board. For three or fewer ports, the standard COM1, COM2 and COM3 ports were used. When installed, the Digiboard used different addresses for COM3 and COM4 (along with special addresses for COM5 to COM10) and the software had to adapt to the two hardware configurations.

To simplify the serial port interconnect, handshaking lines were not used (transitions were ignored). Only transmit data, receive data and signal ground are required.

3. Software Design

3.1 Introduction

The following sections provide the details of the communications software design as well as the implementation. The software is contained in two different files: COM.C contains the C language routines that provide high and low-level communications, and SERIAL.ASM contains all the real-time routines that provide basic interrupt-driven services for the hardware. First the real-time software will be discussed followed by low-level and high-level communications services.

3.2 Real-time Software

DOS (Disk Operating System) does not provide interrupt driven communications through the serial ports. The only way to have the necessary control and response time for the communications software was to provide interrupt driven communications in assembly language. Once interrupts proved necessary for serial ports, a further requirement to ensure that interrupts were tidied up prior to exit forced the use of critical event trapping (control-C presses and critical error exits). As well, timeouts required for the high-level protocols necessitate interrupt driven timer routines. These routines were written to provide the minimum required service with a fast response time (more sophisticated service is to be provided by high-level language routines). SERIAL.ASM contains all of the real-time services written in assembly language.

3.2.1 Serial Ports

To ensure rapid response, interrupt driven communications were used. [2] was used as the basis for a single-port interrupt service routine. There were several small bugs in the code shown in [2] which had to be corrected. To provide service for multiple serial ports, it was necessary to extend the interrupt service routine. In addition to separate buffers with pointers, separate settings for the ports and separate status flags, it was also necessary to service the different IRQs (interrupt request) used. A further complication entered because there were two possible types of hardware that used different addresses and IRQs for COM3 and COM4.

All services provided are C-callable. They include setup and restoration of the interrupts, configuration of the serial ports, reading and writing to the serial ports and getting the composite status of the serial ports. More internal details are provided for each service and the service routine below.

3.2.1.1 Open Serial Ports

Each call to *open_ser* opens one serial port. The routine first checks the board type parameter to see if Digiboard or standard addresses are in use. In the latter case, the IRQ number and port address table used for setting up serial ports are modified (from the Digiboard defaults) to reflect the standard values. At this stage, all interrupts are disabled until vector manipulation is complete at the end of this routine. The routine then checks to see if the port has already been opened - if so, an error is generated and the routine returns. The serial port hardware is then cleared and initialized. Next the routine checks to see if the interrupt is already in use (each IRQ could have multiple serial ports using it) - if not, the interrupt vector is setup. Finally, the interrupt controller is reset and interrupts are re-enabled.

Configuring the serial port is then accomplished using the routine *set_ser*. This routine is used to configure a serial port's baud rate, bits/character, stop bits and parity. The four characteristics are combined into one 8-bit configuration byte. When invoked, this routine breaks up the configuration byte to load up the hardware registers.

3.2.1.2 Close Serial Ports

A call to *close_ser* closes one serial port. If the port was not opened, then this routine returns immediately with no error. When the port is open, this routine disables the serial port hardware and then checks to see if any other port is using the IRQ. If not, then the vectors are restored to their original values.

3.2.1.3 Composite Status of the Serial Ports

The composite status of all the serial ports is available using the routine *stat_ser*. This status has several bits that report problems with the serial ports. They include: interrupt called but no serial port generated the interrupt, a RS-232 handshaking line changed state despite this interrupt being disabled, a UART (universal asynchronous receiver/transmitter) error or break occurred despite being disabled, receive and transmit buffer overflows and finally transmit buffer not empty. The last three bits are composite status in that they represent the "OR" of the states of all of the active ports. In other words, if one of these bits is set then at least one of the serial ports had the associated problem.

3.2.1.4 Receiving Data from Serial Ports

Data received is stored in the receive ring buffer by the interrupt service routine. Upon being called by a C program, *read_ser* first compares the get and put pointers to determine if there are any characters in the receive ring buffer (if there are no characters then the routine does an error return). When there is data, the next character is removed from the ring buffer and returned to the calling routine.

3.2.1.5 Transmitting Data Out of the Serial Ports

When the routine *write_ser* is called to send a character out of a serial port, the transmit ring buffer is checked to see if any characters are still queued. If so, or if the transmitter is not ready, then the current character is added to the buffer which will be emptied one character at a time upon transmit buffer empty interrupts. When saving the current character in the transmit ring buffer, the routine also checks to see if the buffer is full - in which case the transmit buffer overflow bit is set in the composite status. If the ring buffer is empty and the transmitter is ready, then the character is sent right away to the serial port.

3.2.1.6 Serial Port Interrupt Service Routine

The serial port interrupt service routine handles both IRQ3 and IRQ4, the two interrupts used by serial ports. Within the interrupt service routine, there are four types of interrupts serviced: control line change, transmit buffer empty, receive character available, and break/UART error event. Of these, control line change and break/UART error should not occur (because they should be masked) and are serviced by clearing the interrupt and setting the appropriate error bit in the composite status.

The service routine is only invoked by a serial port event - it is never called by another routine. Upon being invoked, *ser_int* first saves all the current context by pushing all the registers that it uses on the stack. The service routine examines all the in-use serial ports and services any of them that have the interrupt bit set. This means at least one serial port is serviced but not more than the number being used. If no in-use serial ports are found with their interrupt bit set, then the service routine sets the invalid interrupt bit of the composite status and exits. Once an in-use port with the interrupt bit set is found, the interrupt identification register is used as an offset for a jump table to the appropriate interrupt type.

For transmit buffer empty interrupts, the service routines checks for characters available in the transmit ring buffer. If available, one character is sent out the serial port. Otherwise, no action is taken.

For receive character available, the service routine first ensures that there is space available in the receive ring buffer. If not, the receive buffer overflow bit is set in the composite status. When there is space, the character is added to the receive ring buffer.

Prior to returning from the interrupt, the interrupt controller (as distinct from the serial port hardware) is given the appropriate command to clear the interrupt or interrupts that occurred. As noted before, the interrupt service routine, once invoked, services all used serial ports that have an interrupt condition. Then the context is restored by popping the used registers from the stack.

3.2.2 Control-C/control-break Handler

DOS normally handles control-C and control-break keypresses by aborting the program, closing open files and then returning to the DOS prompt. DOS does not restore most interrupt vectors as part of this operation, so DOS is likely to crash if a program using interrupts is allowed to be aborted by control-C or control-break. It is necessary for the user software to be able to trap these keypresses. The hearts of the control-C and control-break handlers (*break_int* and *ctlc_int*) were taken from [2]. Once either keypress occurs, the software sets a flag indicating that a control-C or control-break was pressed. The user software check this flag by making periodic calls to *press_break*. The user software can either ignore the keypress or can restore interrupts followed by an exit. C-callable routines are supplied (*open_break* and *close_break*) that trap these keypresses and restore the DOS handler.

3.2.3 Critical Error Handler

Critical errors are severe errors that occur with the peripherals of the computer (such as the floppy disk drive or printer). One example of a critical error is trying to read a floppy disk when there is no disk in the drive. When a critical error occurs, DOS provides the standard prompt describing the critical error and allowing the user to specify the action "Abort, Retry, Ignore or Fail." If the user specifies "Abort", the program is aborted and control returns to the DOS prompt. Unfortunately, there is no user abort routine to allow interrupts to be restored prior to returning to the prompt, so DOS will likely fail at this point. The user software must trap the critical errors and service them; if "Abort" is chosen, then the user software must restore the interrupts prior to returning control to DOS.

The critical error handler (*crit_hand*) was only slightly modified from the one given in [2]. Upon critical error, the user is prompted with a non-specific "Critical Error Occurred: Abort, Retry, Ignore, Fail?". If the user chooses "Abort", then all the interrupts are restored through hard coded calls to the appropriate close routines. Once this is done, control is returned to DOS to finish the abort processing. If any other value is chosen, then control is returned to DOS for finish the appropriate processing (for

example upon user selecting "Retry" then DOS retries the operation) and once the operation is complete, DOS returns control to the user software (but not for "Abort").

C-callable services are provided for setup and restoration (*open_crit* and *close_crit*) of the critical error handler. If software is written that uses any other interrupt, then changes must be made to the critical error handler. The appropriate close must be added at the end of the critical error handler which must then be reassembled.

3.2.4 Timers

Stop-and-wait ARQ requires the ability to wait a period of time after a message is sent before it is declared lost and retransmitted. To provide this facility, a timer interrupt service routine was written. Upon interrupt, the routine decrements all the timers once until they have reached zero. The DOS 16.7 Hz timer interrupt was redirected to this timer interrupt service routine. A separate routine examines the remaining count to check for expiry of a timer.

The routines provided are C-callable and allow setup and restoration of the timer interrupt vector (*open_time* and *close_time*) as well as routines to set the individual timers (*set_time*) and to check them for expiry (*chk_time*). *chk_time* actually returns the remaining count (which is zero on expiry). The timer number used matches the serial port number used. Since there is no COM0, timer 0 is extra and can be used in the user software as a general purpose count-down timer.

3.3 Low-level Communications

Low-level communications are provided by the routines *getc_low*, *gets_low*, *putc_low*, and *puts_low* that get or put characters or strings to the serial ports. Each of these routines, when called, first determines the serial port that matches the low-level station. *putc_low* and *puts_low* send out the character or string using calls to *write_ser* (described previously in section 3.2.1.5). *gets_low*, using calls to *read_ser*, retrieves characters and puts them in a holding buffer until the specified terminator is reached. If the terminator is not yet reached and there are no characters available, the routine returns a status value that indicates that a string is not yet available. A later call will finally retrieve the remaining characters (including the terminator) and return them to the calling routine. The routine *getc_low*, first checks this holding buffer for characters - if found, a character is removed from the holding buffer and returned. If the holding buffer is empty, the routine uses *read_ser* to get a character. The routine returns this character or no data available.

3.4 High-level Communications

This section describes some of the details of the high-level communications software. First, enabling and disabling communications will be examined, then the software involving receipt and transmission of high-level messages will be described. Finally, some of the important variables and data structures will be detailed.

3.4.1 Enabling and Disabling Communications

The routine *open_com* is used to enable high and low-level communications. First the data structures are initialized and the configuration file is read using the internal routine *read_config*. This

internal routine opens and reads the configuration file, setting up the serial port data structures as each link declaration is processed. Once *open_com* enables the critical error handler, control-C/control-break handler and the timers, all the serial ports declared in the configuration file are opened using a separate *open_ser* for each link. Finally, the serial port parameters obtained from the configuration file are used to set up the serial port hardware using calls to *set_ser*.

The routine *close_com* closes all the serial ports using calls to *close_ser* and then disables the timers. Finally, the DOS handlers for control-C/control-break and the critical error are restored.

3.4.2 **Receiving Messages**

Messages are received by calls to *get_com* which first checks for any control-C/break keypresses or too many errors (total or by link) and returns if either of these are detected. Otherwise, *get_com* then calls the internal routine *getmess* once for every active high-level port. *getmess* moves characters from the ring buffer, via calls to *getline*, which in turn calls the real-time routine *read_ser*, and places them into the receive message buffer. Characters are removed up until the message terminator is received. The resultant string is classified as short (for control messages) or long (for user data). Long strings are then checked for header integrity and the checksum is verified. This results in the message being classified as one of: valid message, bad message, Ack or Nak. The Ack is further verified to ensure that it is appropriate for the transmitted message, if not, it is declared to be an invalid Ack. The class of message received then serves as the input for transitions in the receiver state machine. The next sections will detail the receiver state machine and each of the possible states.

3.4.2.1 Receiver State Machine

Fig. 3. shows the receiver state diagram for high-level protocol. There are four possible states shown by the filled-in circles. The arrows show the state transitions which occur normally as a result of received data. Sending a user message or obtaining a receiver timeout can also cause state transitions. The reason for the transition is shown in bold whereas italics are used for the action taken on transition.

3.4.2.2 Ready State

The Ready state is the most commonly used state in the receiver. This is the start-up state and the state used while waiting for messages. As long as valid messages are received (and none sent) the receiver stays in this state. There are only two ways to leave this state. If an invalid (corrupted) message is received in the Ready state, a Nak is sent and the receiver changes to the Nak Sent state. The transition to the Message Sent state occurs, not through the received data, but through the transmitter when a message is transmitted.

3.4.2.3 Nak Sent State

The Nak Sent state is distinguished from the Ready state by the timeout. On timeout, the Nak is retransmitted and the timeout is restarted. On receipt of a valid message, the receiver returns to the Ready State. If further corrupted messages are received, the Nak is retransmitted and the state does not change.

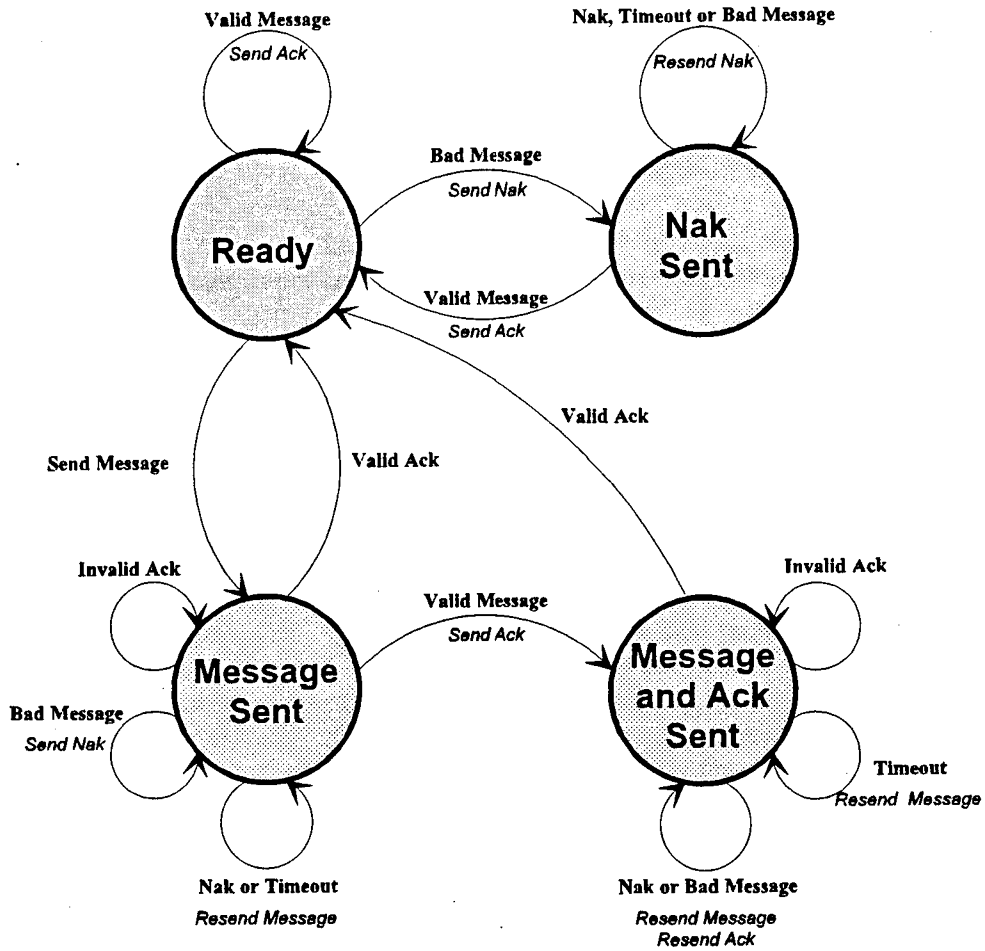

Fig. 3. High-level protocol receiver state diagram.

3.4.2.4 Message Sent State

The Message Sent state is entered by the user transmitting a message. Message transmission is only permitted when the receiver is in the Ready state. Upon transmission, the receiver is put in the Message Sent state. While in this state, a timeout waiting for the Ack is set. Upon receipt of a Nak or on expiry of the timeout, the transmit message is resent and the timeout restarted. If a valid message is received in this state, the transition to the Message and Ack Sent state occurs.

3.4.2.5 Message and Ack Sent State

The Message and Ack Sent state is an infrequently used state. To get into this state, a message must be transmitted and another valid one received and acknowledged prior to the Ack of the transmitted message. In this state, there is ambiguity if a Nak is received - it is not possible to know if the Nak is in response to a problem with the acknowledgement or with the original message (which could have been lost). In the case that a Nak is received, both the Ack and the transmit message are resent - resulting in at least one duplication at the far end, but no losses. This state functions otherwise as the Message Sent state.

3.4.3 **Sending Messages**

Messages are sent using the routine *send_com* which frames the message, sets the checksum and then checks to see if the receiver is in the Ready state (which ensures all previous messages have been successfully transmitted). If so, the routine *sendstr* is used to send the string using calls to the real-time routine *write_ser*. Also the countdown timer is started for the timeout using *set_time* and the receiver state is changed from Ready to Message Sent.

3.4.4 **Internal Data Variables**

3.4.4.1 Station Numbers

The number used internally for the stations is based upon the definitions given in the COM.H file. Each high-level station is assigned a fixed number within the range: 1 up to but not including LOW_BASE. A value of 0 is used to indicate a bad station. Any value greater or equal to LOW_BASE is the station number for a station on a low-level link. Low-level stations are the sum of LOW_BASE and an index. This index corresponds to the order that the low-level link declarations occur in the configuration file (0 is the index for the first low-level link).

3.4.4.2 Serial Port Numbers

The values used for serial port numbers internally correspond to the associated COM port number. Therefore, the serial port number for COM2 is 2. The range is 1 to 10.

3.4.4.3 Message Numbers

The message numbering scheme involves only two numbers 0 and 1. They correspond in the message frame to 'h' and 'H' respectively. For acknowledgments, the numbers correspond to 'ack' and 'ACK' respectively.

3.4.4.4 Active Port Structure - s

The structure s details the active links for both high and low-level communications. It is indexed by position in the configuration file and has one member for each link. For each link, the following information is stored: the station number at the far end of the link, the serial port number and the serial port settings (such as baud rate).

3.4.4.5 Serial Port Structure - **p**

The structure **p** details the serial ports and is indexed by the serial port number (1 to 10). This structure only contains useful information for serial ports used in high-level communications links. For each serial port, the following information is stored:

- state of the receiver
- station number at the far end of the link
- number of consecutive errors
- maximum allowable number of consecutive errors
- number of ticks before timeout

- message number expected for the next receive message
- pointer for the receiver buffer
- holding buffer for the receiver
- previous received message string (for duplicate message detection)
- message number for the next transmit message
- previously transmitted message string (for retransmission)

4. Testing

4.1 Method

The development of the communications software required the use of multiple stations. Initially, one end of the link was the development computer and the other was the HP 4952A Protocol Analyzer. The analyzer was set up to send messages and also to respond with acknowledgments to messages sent from the computer.

Once the software was basically working, two computers were connected each running an early version of the program SER_DEMO (given as the example program in the Communications Software User's Guide found in Appendix A). This program reports all messages received and any communications errors. It also generates messages at the press of a key. The next step in the testing was to connect three computers together and send messages to one computer at the same time. No problems were found.

Practical testing was done during verification of the beacon monitoring and data logging software - where the communications software was integrated with user programs. The Beacon & Reference Monitor, monitoring the satellite beacon, was configured to send the measurement results routinely to the Data Logger. An overnight run was conducted to test the RF hardware and the two computers with their associated software. This test highlighted some problems with the initial version of the communications software and its usage.

4.2 Problems Discovered

There were times during the testing where multiple communications errors occurred followed by an exit when too many errors were counted. The problem turned out to be with the Beacon & Reference Monitor which was a slower AT-class computer. This computer did not have the processing power necessary to service all the communications at 9600 baud at the same time as performing its primary function. By reducing the baud rate to 2400, this problem was alleviated. This could have also been rectified by replacing the AT-class machine with a 386 or 486 computer.

Another problem with communications was discovered where both lost messages and duplicate messages were occurring. It turned out that several of the measurements done by the Beacon & Reference Monitor over GPIB (general purpose instrument bus) were taking as long as 15 seconds (during which there could be no calls to *get_com* to process the handshaking). This was fixed by extending the timeout period for the link to 30 seconds at both the Data Logger and the Beacon & Reference Monitor.

Later, during the trials, the Data Logger occasionally stopped servicing one of the links. This turned out to be a problem with the interrupt service routine. The same interrupt service routine is invoked for all links and it was coded to look only for the first link needing service. This caused a conflict when more than one source of interrupt occurred simultaneously (the Data Logger had a large number of links). To correct this problem, the interrupt service routine was modified to ensure that all links (not just the first) that needed servicing were serviced.

4.3 Usage Problems

During integration prior to the trials, two usage problems were brought to light. They were sufficiently common that future versions of the software should try to alleviate or at least provide notification of these problems.

The first problem was an insufficient number of calls to *get_com* which processes the messages. This resulted in messages or acknowledgements being lost and later duplicated. The root of the problem was usually a time critical area in the user software that was waiting for some other hardware event. It was very easy for the user to create a program with a loop waiting for a certain bit to be set without calling *get_com* within this loop. If this waiting period was longer than the timeout, a problem occurred. The solution to this problem was to ensure that *get_com* was called in all waiting loops.

The other problem resulted in general communications or framing errors on a link. This was caused by the user including carriage returns and linefeeds in the message itself (this often occurred when the same message sent to the Data Logger was also sent to the local computer display which requires the linefeed). The linefeed would cause a premature detection of the end of message. This problem could also occur when other control characters are embedded in the message because these characters are discarded at the receiver prior to computing the checksum (which would then fail).

4.4 Results

After correcting the problems within the communications software found prior to and during the trials, and correcting the problems in the user software, the communications software performed successfully for the rest of the trials. Both the high and low-level communications provided the necessary services for the users to allow communications among the distributed processors and to allow control specific hardware devices. During these trials, the communications software serviced 8 high-level interprocessor links and 3 low-level computer to instrument links.

It should be noted that AT-class machines cannot run high-level communications at 9600 baud or faster because of processing limitations inherent in these slow machines. 386 and 486-based machines can handle multiple links at 9600 baud without problems and are better suited to the tasks required for the Skynet EHF Trials.

5. Conclusions

5.1 Summary

The Skynet EHF Trials involved multiple computers which had to intercommunicate. The communications software presented in the previous chapters provided the communications services necessary for the distributed processing used in these trials. The challenge was to develop a system that was easy to integrate with the user software as well as to ensure that the communications hardware and software did not conflict with special purpose boards in the various computers.

For simplicity, stop-and-wait ARQ protocol was used for high-level message passing. This provided robust message handling and error-free transmissions. To simplify debugging, but at the expense of efficiency, only printable characters were used for the messages and framing. Also, low-level communications services that do not require handshaking were provided for equipment control. The software was developed in the C language with the real-time hardware interface portion written in assembly language.

The communications software presented met the challenge and, after extensive testing, was proven to provide the necessary communications among all the processors and special devices.

5.2 Future Work

In hindsight, improvements could be made to the communications software in three main areas: detection of usage problems, flexibility and better software approaches. The following sections describe these areas in more detail.

5.2.1 Detection of Usage Problems

Carriage returns, linefeeds or other control characters in a high-level message should be detected prior to attempting to send the message. This could be done simply at the start of *send_com*, and if control characters are detected in the string, there should be an error return from *send_com*.

The time between calls to *get_com* could be monitored by the extra timeout counter (timer 0 is available) to ensure that long periods between calls to *get_com* are reported right away. This timer should be set for a timeout period of one-tenth of the smallest timeout for all links (or possibly to a user specified value from the configuration file). When *get_com* is called and this timer has expired, an error message should be given such as "The time between calls to *get_com* is too long." This timer would be restarted at each call to *get_com*.

5.2.2 Flexibility

The current communication software specifies, in the header file COM.H, the valid long and short station names. This system worked for the Skynet EHF Trials because the names did not change. If it is desired to have a different configuration, then the header file must be changed and the user and communications software must be recompiled. It would be more flexible if the valid station names were contained in some type of setup file and read at execution time. In this case, all stations must have the same setup file.

5.2.3 Better Approaches

Certain aspects of the program were designed early on in the development stage and proved to be cumbersome or cryptic later. The first instance of this is the composite status for the real-time serial port routines. This status returns only the combined status of all ports when an individual port status would be more useful. This is most important for status items such as buffer overflows. The other aspect of the status is that it was never used by the high-level communications software. This status should be examined each time *get_com* is invoked and if necessary the error message should be returned. Also, for low-level communications the status should be checked before sending data to ensure there is room in the buffer.

The last problem is the method of generating message numbers are used for messages and acknowledgements. The method of using the case of the letters to indicate the message number is cryptic. It would be better to have a message number field and to include message number with the acknowledgment.

Appendix A

Communications Software User's Guide

1. Introduction

This appendix describes the use of the communications software. First high-level then low-level communications are covered. Next the serial port configuration file used by the communications software is documented. Finally a programming example using high-level communications is provided. The interface details of each of the communications software routines are given in Appendix B: Communications Software Programmer's Reference.

2. High-level Communications

High-level asynchronous serial communications involve robust message handling with confirmation of reception at the far end of the link. The handshaking is handled by the software - the user is only responsible for specifying the destination, message type and message data. The following sections will detail the information necessary to send a message as well as the information available on receipt of a message. Then the communications errors and communications termination will be detailed.

2.1 Enabling and Disabling High-level Communications

High-level asynchronous serial communications (as well as low-level serial communications) are enabled by the routine *open_com*. This routine reads the configuration file, sets up the message handling routines and takes over the serial ports specified. No communications can occur until this routine is called. It is only necessary to call it once regardless of the number of links in the configuration file.

Prior to termination of the user program, it is important that the routine *close_com* be invoked to remove all the message handling routines and to free up the serial ports. If this routine is not invoked, the computer will likely hang upon exit from the user program.

2.2 Sending Messages

To send a high-level message, one uses the routine *send_com* along with several parameters: destination station number, message type number and message data. The destination station numbers are defined in COM.H. Keywords for valid station numbers are:

DATA_LOGGER	Data Logger & Experiment Controller
BEACON_MON	Beacon & Reference Monitor
BURST_DEMOD	Burst DPSK Demodulator Host
TX_PROC	CRC Transmit Processor
EPHEM_PROC	Ephemeris Processor
SYNC_PROC	Synchronization Processor
CRC_ANTENNA	CRC Antenna Controller Host
T85_ANTENNA	T85 Antenna Controller Host

The station number can also be obtained from the routine *look_com* by giving the long station name as a string.

The message type numbers are defined in COM.H and specify which type of message is to be sent. The message type is distinct from the message data which contains a string. Keywords for message type numbers must be one of the following:

COMMAND	Command message, used to start/stop another processor or request status
CONFIGURE	Configuration message, to choose setup or process for another processor
LOG	Log message, to be stored in the log file
STATUS	Status message, response to command (if necessary)
POINT	Initial antenna pointing information, generated by the ephemeris processor
MOD_POINT	Modified antenna pointing information, modified by the sync processor
TIME_STAMP	Time of day message, time of day distributed by the logger
ERROR	Error condition message, error to be stored in the log file

The message types and any associated responses used must be agreed upon by the two stations on the link. For example, the Sync Processor would send a Command message to the Tx Processor to initiate a certain type of transmit waveform. The Tx Processor would respond with a Status message to indicated that the transmit waveform was now valid.

Message data consists of a variable length string, formatted as specified by the experiment and is an optional parameter. If there is no data, a null string should be passed to the routine.

2.3 Receiving Messages

Messages are obtained by the routine *get_com* with a return of VALID_MSG. This routine also handles the handshaking, so it must be called repeatedly. If the routine is not called after a message comes in, there will be no handshaking and a timeout error will be generated at the other end of the link.

When a message is received, the message type, message data and the source station are returned by this routine. The message type and valid stations were shown in the previous section. The message data is contained in a null-terminated string and in the event of no message data, the string will be a null string.

2.4 Communication Errors

Communication errors such as lost messages are reported in *get_com* using the COMM_ERR return value. The return parameters provide the communications error number, the station at the far end of the link that had the communication error and the error text. See the Communications Software Programmer's Reference in Appendix B for more details of the C program interface. The following table provides details for each error including likely causes and remedies.

Note that there should not be any errors in normal operation. Using proper connectors and keeping the line lengths within the RS-232 standard should provide error-free transmissions. If errors do occur, it is usually an indication that something is wrong with the hardware setup.

Err No	COM.H Define	Error Text	Cause	Remedy
1	CPTACK	Ack corrupted	A nak was received in response to the previously transmitted ack	- Check timeout and *get_com* call frequency - Check connections
2	CPTNAK	Nak corrupted	A nak was received in response to the previously transmitted nak	- Check timeout and *get_com* call frequency - Check connections
3	CPTRXA	Receive message or ack/nak corrupted	An unrecognizable string was received May be one of: - errors in framing - bad checksum - from station does not exist or is the wrong one - to station does not exist or is the wrong one - message type is invalid - garbage on the line	- Check timeout and *get_com* call frequency - Ensure there are no control characters in the message strings (especially '\n','\r') - Verify station names in configuration file - Check connections
4	CPTTXA	Transmit message or ack corrupted	A nak was received after both and ack and a message were transmitted (in response to either one)	- Check timeout and *get_com* call frequency - Check connections
5	CPTTXM	Transmit message corrupted	A nak was received in response to the previously transmitted message	- Check timeout and *get_com* call frequency - Check connections
6	EXTACK	Extra ack received	An ack was received when none was needed	- Check timeout and *get_com* call frequency
10	LSTACK	Ack lost, duplicate message	The latest receive message number is out of sync with the expected message number and the message is the same as the previous one - this is a duplicate message	- Check timeout and *get_com* call frequency on the other end of the link
11	LSTNAK	Nak lost	A nak was sent and no response was received prior to timeout	- Check timeout and *get_com* call frequency on the other end of the link
12	LSTRXM	Receive message lost	The latest received message number is out of sync with the expected message number and the message is different from the previous one - a message must have been missed	- Check timeout and *get_com* call frequency on local station
13	LSTTXM	Transmit message lost	A message was sent and no response was received prior to timeout	- Check timeout and *get_com* call frequency on the other end of the link

The most common source of problems is the frequency with which calls are made to *get_com*. Since this routine provides all the handshaking, if it is not called often enough, then messages are not acknowledged within the timeout period of the sending station. The routine *get_com* does not require a lot of processing power enabling the user to call it frequently with minimal effect on the primary task

of the computer. For more details on *get_com*, see the Communication Software Programmer's Reference in Appendix B.

A related problem is when the host computer does not have sufficient processing power to service the serial ports at full speed. In that case, the solution is to lower the baud rate of the serial ports, reduce the number or length of messages, and to minimize the number of ports to be serviced concurrently.

The next most common source of problems is the use of control characters in the message string. Since the high-level protocol framing uses control characters to denote end-of-message, the incorporation of control characters in the user string will cause the protocol to terminate prematurely the receive message. To send a two-line message, first split it into two one-line messages and send them with two separate calls to *send_com*.

2.5 Termination

The routine *get_com* can also request program termination by the returning of QUIT. The termination type and sometimes the originator number are available. Keywords for the termination types are:

TOTAL	Too many total errors occurred (sum of all errors on all links)
CONSEC	Too many consecutive errors on any one link (the originator specifies which link had too many errors)
BREAK	Control-C or control-break was pressed

The user software can ignore this request, but with either of the communications error terminations, high-level communications is no longer effective because it is continuously tied up reporting errors. The routine *flush_com* may be used to reset a link after too many consecutive errors, but should only be called once the reason for the errors is removed. The control-C/control-break keypress can be used to exit the program or the user software can ignore these keys if an user initiated abort is not desired.

Another source of termination which is beyond user software control, is the Abort selection upon a critical error. Critical errors are operating system errors such as no floppy disk in the drive when trying to read a directory. Because the operating system does not return control to the user software upon the selection of Abort (as opposed to Retry, Ignore or Fail), these critical errors are trapped by the communications software. There, a simplified critical error handler checks for the Abort response and if selected, does the equivalent of *close_com* automatically prior to the return to DOS.

3. Low-level Communications

Low-level communications involve the sending and receiving of individual characters or character strings. There is no handshaking, error control or flow control. It is meant primarily for controlling peripherals (such as an antenna controller) using the serial ports. Low-level communication routines were added to the communications software package because direct programming of the serial ports would conflict with high-level communications controlling of the serial port interrupts. The following sections detail the enabling and disabling of low-level communications, sending data, receiving data and termination.

3.1 Enabling and Disabling Low-level Communications

Low-level communications (as well as high-level communications) are enabled by the routine *open_com*. This routine reads the configuration file and sets up the serial ports as specified. No communications can occur until this routine is called and it is only necessary to call this routine once regardless of the number of links in the configuration file. The routine *close_com* must be called prior to termination to free up the serial ports. If this routine is not invoked, the computer will likely hang upon exit from the user program.

3.2 Sending Data

To send single characters out a serial port, the routine *putc_low* should be used. This routine will send any one character out the serial port. If it is desired to send a string, the routine *puts_low* can send a null-terminated string. If it is necessary to send a null as part of a string, then the string should be broken down into string, null character and string. These then should be sent out using calls to *puts_low*, *putc_low* and *puts_low* respectively.

3.3 Receiving Data

Single characters can be received from the serial port using the routine *getc_low*. This routine will obtain the next character from the ring buffer regardless of value. To obtain a terminated string from a serial port, the routine *gets_low* can be used. This routine allows the user to specify the string terminator and then retrieves all characters up to (but excluding) the specified terminator. The string terminator cannot occur within the string.

3.4 Low-level Termination

The routine *get_com*, while normally only used for high-level communications, can be used to detect user termination requests via control-C and control-break keypresses. All other features of *get_com* are not used for low-level communications. The only possible returns are NO_MESSAGE (no keypresses) and QUIT (termination request). The parameter associated with QUIT can have only one value: BREAK to indicate that control-C or control-break has been pressed. The other values for this parameter can only occur in high-level communications.

The user software can ignore this termination request with no consequences to the communications software, but it is better to respond to the users attempt to exit the program. Prior to termination of the program, it is important that *close_com* be invoked to restore interrupt vectors.

Another source of termination, beyond the user software control, is an Abort selection by the user in response to a critical error. Critical errors are operating system errors (such as no floppy disk in drive or printer not ready). Because the operating system does not return control to the user software upon the selection of Abort (but it does for Retry, Ignore or Fail) these critical errors are trapped by the communications software. There, a simplified critical error handler checks for the Abort response and, if selected, does the equivalent of *close_com* prior to the return to DOS.

4. Serial Port Configuration File

This file contains the declarations necessary to specify completely all the communications links for the local computer including all connected stations. It is read once at the start of the program and cannot be changed while the program is running. SERIAL.CFG is the default name for this file, but another filename can be specified using the routine *config_com*.

The configuration file is an ASCII text file, that can be edited using any text editor. Case is unimportant. Blank lines and comment lines (any line starting with an ";") are ignored. Leading or trailing tabs and spaces are ignored, but cannot occur inside keywords or values. The configuration file consists of keywords (and their associated values), comments and blank lines. The following are valid keywords:

Keyword	Declaration Type	Description
FROM	Local Station	Local station name
BOARD_TYPE	Local Station	Serial board type
MAX_ERROR	Local Station	Maximum total errors for abort
TO	Link	High-level link connected station name
LOW_LEVEL	Link	Low-level link connected station name
BAUD	Link	Baud rate
BITS	Link	Number of bits per character
CONSECUTIVE	Link	Consecutive errors for abort
PARITY	Link	Parity type
PORT	Link	COM number
STOP	Link	Number of stop bits

The order of the keywords is important within the file. The local station declaration must precede any link declarations. Within the link declarations (and after the link connected station name) any order can be used for the link parameters (such as baud rate and parity). The Local Station Declaration defines the local station and thus cannot be omitted. The link declarations define communications links to various other computers or serial devices. There can be no, one or up to ten link declarations. The serial port configuration file must have the following form:

Local Station Declaration
Link Declaration
Link Declaration
⋮

4.1 Local Station Declaration

The local station declaration defines the local station, specifies the serial board type and sets the maximum number of communication errors before aborting. The keywords used are FROM, BOARD_TYPE and MAX_ERROR. The format for the declaration is:

Local Station Name
Local Station Parameters

4.1.1 Local Station Name (FROM)

The local station must be named as one of the predefined computers (Data Logger & Experiment Controller, Beacon & Reference Monitor, Burst DPSK Demodulator Host, CRC Transmit Processor, Ephemeris Processor, Synchronization Processor, CRC Antenna Controller Host or T85 Antenna Controller Host.) This line must be the first line of the Local Station Declaration and hence will be the first (non-comment) line in the file. There can only be one local station, so there is only one such declaration allowed. This declaration cannot be omitted. The format of this declaration is given below:

FROM={DATA_LOGGER | BEACON_MON | BURST_DEMOD | TX_PROC | EPHEM_PROC | SYNC_PROC | CRC_ANTENNA | T85_ANTENNA}

4.1.2 Local Station Parameters

The local station can be qualified by two parameters: the type of serial board used and the maximum number of errors before aborting. Both of the parameters have defaults and can be omitted. The order of the parameters is unimportant.

4.1.2.1 Serial Board Type (BOARD_TYPE)

The Digiboard Digichannel PC/8 eight-port serial board was used on most computers. This board had slightly different characteristics for the use of COM3 and COM4 compared to standard PC serial ports. This declaration allows the board type to be specified (default is the Digiboard).

BOARD_TYPE={STANDARD | DIGIBOARD}

4.1.2.2 Maximum Number of Errors (MAX_ERROR)

If the total number of communication errors received from the links exceeds the maximum number of errors, the communications software causes the program to abort. This ensures that software or hardware problems are recognized and can be acted upon. In normal operations, there should be no communication errors. This value, number_errors, must be greater than 0 and less than 30000. The default value is 100.

MAX_ERROR=number_errors

4.2 Link Declaration

The link declaration consists of several lines describing the connected station and the parameters of the serial link. Included are the keywords TO, LOW_LEVEL, BAUD, BITS, PARITY, PORT, STOP and CONSECUTIVE. There can be from zero to ten link declarations. The format for link declarations are:

Connected Station Declaration
Link Parameters

4.2.1 Connected Station Declaration

There are two types of links: high-level links involving robust message handling between computers, and low-level links for a computer to drive a serial device such as a clock or antenna controller. Either type of declaration must precede all of the associated serial port parameter declarations. Succeeding connected station declarations are treated as separate links.

4.2.1.1 High-level Connected Station Name (TO)

For high-level communications this connected station declaration must be used. The declaration defines the computer at the far end of the link (Data Logger & Experiment Controller, Beacon & Reference Monitor, Burst DPSK Demodulator Host, CRC Transmit Processor, Ephemeris Processor, Synchronization Processor, CRC Antenna Controller Host or T85 Antenna Controller Host.) The format of the declaration is given below:

TO={DATA_LOGGER | BEACON_MON | BURST_DEMOD | TX_PROC | EPHEM_PROC | SYNC_PROC | CRC_ANTENNA | T85_ANTENNA}

4.2.1.2 Low-level Connected Station Name (LOW_LEVEL)

If robust message handling is not desired, low-level links can be created to support communications with serial devices. This declaration defines a reference name for the far end of the link that is used later for low-level communications routines. The reference name given must be unique. The format of the declaration is given below:

LOW_LEVEL=reference_name

4.2.2 Link Parameters

These declarations define the serial port to be used and specify the parameters for asynchronous communications - including baud rate, parity, number of bits per character, number of stop bits and maximum number of consecutive errors. With the exception of the serial port to be used, all parameters have a default value and are optional. The order of the declarations within this section is not important. Keywords should not be used more than once per link, because the second occurrence overrides the first. This section is finished at end-of-file or where there is subsequent connected station declaration.

4.2.2.1 Baud Rate Declaration (BAUD)

This keyword specifies which of the valid baud rates are to be used for the serial port. It is an optional declaration and if it is not present, the baud rate defaults to 9600.

BAUD={110 | 150 | 300 | 600 | 1200 | 2400 | 4800 | 9600}

4.2.2.2 Bits Per Character Declaration (BITS)

This declaration controls the number of bits per character for asynchronous serial communications. The default value is 8 bits per character. This declaration is optional.

BITS={5 | 6 | 7 | 8}

4.2.2.3 Maximum Number of Consecutive Errors Declaration (CONSECUTIVE)

This declaration defines the maximum number of consecutive errors on the link. This is the number of errors that occur in a row without any intervening valid messages. In normal operation, there should be no errors. An abort caused by too many consecutive errors is usually indicative of a hardware fault on the line or that the software at the connected station is not operating properly. The number of errors, number_errors, must be between 1 and 10000. The default value is 10.

CONSECUTIVE=number_errors

4.2.2.4 Parity Declaration (PARITY)

This declaration controls the parity bit, if used. The valid values allow no parity (all bits are data), even parity or odd parity. This declaration is optional and if it is not present, the default value is no parity.

PARITY={NONE | EVEN | ODD}

4.2.2.5 Port Declaration (PORT)

This declaration defines the port to be used and must be present in a link declaration. If it is not present, an error occurs. Each link must use a different serial port, so no two links can have the same port declaration. The valid values include COM ports 1 to 10. In the case of the tenth port, the hexadecimal notation is used giving COMA. AUX is a synonym for COM1.

COM1 and COM2 ports are as defined for normal PCs. The other eight ports use the default address/interrupt definitions of the DigiBoard DigiChannel PC/8 eight-port serial board. (For PC versions of COM3 and COM4 use the BOARD_TYPE declaration.)

The program takes complete control of the serial port declared using the PORT keyword, so it is important that there are no conflicts with the operating system, serial printers, other communication software, networking software or serial mice.

PORT={COM1 | COM2 | COM3 | COM4 | COM5 | COM6 | COM7 | COM8 | COM9 | COMA | AUX}

4.2.2.6 Stop Bits Declaration (STOP)

This declaration defines the number of stop bits transmitted. The selection of 1.5 stop bits is only available when there are five bits per character (1.5 bits is converted to 1 bit for other character lengths and 1 stop bit is converted to 1.5 bits for five bit characters). This declaration is optional and the default value is one stop bit (1.5 stop bits for five bits per character).

STOP={1 | 1.5 | 2}

4.2.2.7 Timeout Declaration (TIMEOUT)

This declaration defines the period to wait before declaring timeout for a high-level link. This is the time that, after sending a message, the sending station waits for the acknowledgement. This time should be greater than the longest period in which the receiving station does not service high-level communications (through calls to *get_com*). The number of seconds for the timeout, timeout_seconds, must be between 1 and 100. The default value is 2 seconds.

TIMEOUT=timeout_seconds

4.3 Sample Configuration File

Below is a sample configuration file for the Burst DPSK Demodulator Host. The local computer is BURST_DEMOD (FROM), the high-level link connected station is the Data Logger and Experiment Controller over COM2 (PORT) at 9600 (BAUD) with 8 bits per character (BITS), no parity (PARITY), one stop bit (STOP), allowing a maximum of 10 (CONSECUTIVE) communication errors in a row and with a timeout 5 seconds (TIMEOUT). A second link allows the computer to control the Comstream Satellite PSK Modem using low-level communications.

```
;=========================================================
;       SERIAL.CFG
;
;       Serial port configuration file for the modem host
;
;---------------------------------------------------------

FROM=BURST_DEMOD
BOARD_TYPE=DIGIBOARD
MAX_ERROR=500

;To Data Logger & Experiment Controller
TO=DATA_LOGGER
        PORT=COM2
        BAUD=9600
        BITS=8
        PARITY=NONE
        STOP=1
        CONSECUTIVE=10
        TIMEOUT=5

;To Comstream Modem
LOW_LEVEL=COMSTREAM
        PORT=COM3
        BAUD=9600
        BITS=8
        PARITY=NONE
        STOP=1
```

4.4 Configuration File Errors

The following table lists all the error that can occur when the configuration file is being read. Also listed are the suggested remedies.

Configuration File Error	Remedy
Board type definition must follow FROM	A BOARD_TYPE definition was found in a link declaration. BOARD_TYPE must be part of the local station declaration.
Cannot open configuration_filename	The configuration file does not exist or is locked.
Comm parameters without TO or LOW_LEVEL	Link parameters are found not preceded by TO or LOW_LEVEL.
Consecutive errors must be in range 1-10000	Ensure number for CONSECUTIVE is within 1 to 10000
Found a definition not preceded by FROM	FROM must be the first keyword in the configuration file
Low-level port name not unique	Two or more LOW_LEVEL declarations used the same name. Choose unique names for each low-level link.
Maximum error must follow FROM	A MAX_ERROR definition was found in a link declaration. MAX_ERROR must be part of the local station declaration.
Maximum errors must be in range 1-30000	Ensure number for MAX_ERROR is within 1 to 30000
Maximum number of ports exceeded	More than 10 link declarations were found. No more than 10 links per computer are supported.
Multiple FROM definition	Only one local station declaration is meaningful.
No FROM definition found	No local station declaration was found. FROM is must be included.
No PORT definition found No PORT definition found for last TO	Link declaration did not include a PORT definition. PORT must be included in each link declaration.
Redefinition of serial port	Link declaration included a PORT definition that has already been used by another link. Each link declaration must have a unique port.
Timeout must be in range 1-100	Ensure number for TIMEOUT is within 1 to 100 (this is in seconds)
Unrecognized baud rate	The number for BAUD was not one of the valid choices. See 4.2.2.1.
Unrecognized bits/character	The number for BITS was not one of 5, 6, 7 or 8.
Unrecognized board type	The value for BOARD_TYPE was not STANDARD or DIGIBOARD.
Unrecognized definition	Unrecognized keyword was found.
Unrecognized FROM station	The value for FROM was not one of the valid choices. See 4.1.1.
Unrecognized parity	The value for PARITY was not one of NONE, EVEN or ODD.
Unrecognized port type	The value for PORT was not one of the valid choices. See 4.2.2.5.
Unrecognized stop bits	The value for STOP was not one of 1, 1.5 or 2.
Unrecognized TO station	The value for TO was not one of the valid choices. See 4.2.1.1.

5. Example Program - SER_DEMO

This section details a program demonstrating the use of the communications software. The program SER_DEMO was used (with minor modifications) to test the high-level communications software and is a useful example of the use of the routines. In the following paragraphs, the program will be detailed, the compiling and linking of the program will be presented and finally the program's listing will be given.

5.1 SER_DEMO Description

The program was first developed to test high-level communications so it includes the ability to report all received messages and the ability to send messages at a keystroke. The program reports all errors and can exit on a keypress.

The main program first starts communications with a call to *open_com*. If any error occurs in the configuration file or setting up of the serial ports, the program exits with the error message "Error in open_com." (This is accomplished using a routine *pabort* which prints out a message, closes the communications using *close_com* and the aborts using *exit*). Once the communications software is started, the program prints out the name of the local station - in the case of the sample configuration file, it would be "burst_demod."

Next the main program looks for a link with the station "data_logger" using *look_com*. If the station is not defined in a high-level declaration within the configuration file, this routine will return an error which is then reported by "Bad station lookup."

The principal portion of SER_DEMO is the loop where keypresses and communications are checked. The routine *checkkey* acts upon keypresses and the routine *checkmsg* checks and displays received messages, communications errors or control-C/control-break termination requests.

5.2 Compiling and Linking SER_DEMO

The software was compiled using Microsoft C 6.0 under DOS 5.0 using the small memory model. The program (and communications software) was compiled and linked using the NMAKE utility. The make file (SER_DEMO.) is given below:

```
ser_demo.exe:   ser_demo.obj com.obj serial.obj
        link ser_demo+com+serial;

ser_demo.obj:   ser_demo.c com.h
        cl /c ser_demo.c

com.obj:        com.c com.h
        cl /c com.c

serial.obj:     serial.asm
        masm serial;
```

5.3 SER_DEMO Listing

```
#include <conio.h>
#include "com.h"

/*
 Local Routines
---------------- */
int checkkey(int mdest);        // Check and action key presses
int checkmsg(void);             // Check for receive messages and others
void pabort(char *msg);         // Print message, close file, and exit



/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - - main- - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

void main(void)
{
    int mlocal;          // Station number of local station
    int ndest;           // Port number for desired destination
    char string[220];    // String buffer used to hold station name

    printf("SER_DEMO V1.1\n");

        // Open all communications
    if ((mlocal=open_com())==BAD_STATION) pabort("Error in open_com");
    printf("Local station is %s\n",stnlstr(mlocal,string));

        // Select the link to the data logger
    if ((ndest=look_com("data_logger"))==BAD_STATION)
            pabort("Bad station lookup");

        // Check for keypress (send messages to 'ndest') and receive messages
    while (checkkey(ndest) == 0) {
        if (checkmsg() != 0) break;
    }
    close_com();
    exit(0);
}


/*=======================================================================*/
/*                                checkkey                               */
/*                                                                       */
/* Description: Checks to see if a key has been pressed and performs the */
/*              necessary action such as sending various messages or exiting */
/*              Control-C/break is not done here, but reported by checkmsg */
/*                                                                       */
/* Returns:     (int)         0 for normal return                       */
/*                            1 for exit from main program due to keypress */
/* In:          (int ndest)   destination station number for messages  */
/* Out:         -                                                        */
/*-----------------------------------------------------------------------*/

int checkkey(int ndest)
{
    int c;              // Character from the keyboard

    if (kbhit() != 0) {                         // Is a key pressed ?
        c = getch();                            // Get the character
        switch (c) {
            case '1':                                   // 1 = Send message 1
                printf("Sending message 1\n");
                send_com(ndest,COMMAND,"Check buffer");
```

```
                break;
            case '2':
                printf("Sending message 2\n");          // 2 = Send message 2
                send_com(ndest,STATUS,"Buffer OK too");
                break;
            case '3':
                printf("Sending message 3\n");          // 3 = Send message 3
                send_com(ndest,STATUS,"Do a third");
                break;
            case '4':
                printf("Sending message 4\n");          // 4 = Send message 4
                send_com(ndest,STATUS,"Quarter");
                break;
            case 'e':
            case 'E':
            case 'q':
            case 'Q':
            case 'x':
            case 'X':
                return 1;                               // e,E,q,Q,x,X = quit
            default:
                break;                                  // Otherwise ignore
        }
    }
    return 0;
}


/*========================================================================*/
/*                              checkmsg                                   */
/*                                                                         */
/* Description: Checks with communications routines for:                  */
/*                      - receive messages from any link                   */
/*                      - communications errors                            */
/*                      - aborts from control-C/break                      */
/*                                                                         */
/* Returns:     (int)           0 for normal return                        */
/*                              1 for exit from main program due to comm   */
/*                                errors or control-C/break                */
/* In:          -                                                          */
/* Out:         -                                                          */
/*------------------------------------------------------------------------*/

int checkmsg(void)
{
    int mstat;                  // Message status - valid, error or quit
    int mtype;                  // Message type number
    int mfrom;                  // Message from station number
    char mdata[220];            // Message data
    char string[220];           // String buffer used to name, type or error

    mstat = get_com(&mtype,&mfrom,mdata);       // Check for message
    if (mstat == VALID_MSG) {                   // Message available
        printf("   from %s ",stnstr(mfrom,string));
        printf("(%s):  \"%s\"\n",messtr(mtype,string),mdata);
    } else if (mstat == COMM_ERR) {             // Communications error
        printf("-- Comm error with %s:  %s\n",stnlstr(mfrom,string),mdata);
    } else if (mstat == QUIT) {                 // End main program
        if (mtype == TOTAL) {                           // Too many errors
            printf("Too many communication errors\n");
        } else if (mtype == CONSEC) {                   // Too many in a row
            printf("Too many consecutive communication errors with %s\n",
                        stnlstr(mfrom,string));
        } else if (mtype == BREAK) {                    // Control-C/Break
            printf("Break detected\n");
        }
        return 1;
    }
    return 0;
```

```
}

/*=========================================================================*/
/*                              pabort                                     */
/*                                                                         */
/* Description: Print error message, close file and abort                  */
/*                                                                         */
/* Returns:     -- no return, aborts --                                    */
/* In:          (char *msg)     Pointer to error message string            */
/* Out:         -- no return, aborts --                                    */
/*-------------------------------------------------------------------------*/

void pabort(char *msg)
{
    printf("%s\n",msg);
    close_com();
    exit(0);
}
```

Appendix B

Communications Software Programmer's Reference

1. Introduction

This appendix provides all the use and interface details for the communications software. The routines are listed alphabetically with the parameters, return values, usage, errors, program fragment providing and example of use and any related routines. The following section provides a functional list of the routines. For more detailed information on use of the whole package, see Communications Software User's Guide in Appendix A.

2. Use of the Routines

All the routine declarations and definitions are made in the header file COM.H which must be included in the user program. The routines were compiled with Microsoft C 6.0 under DOS 5.0 using the small memory model. The C calling convention is used for all routines. The routines can be grouped into three categories: control routines, high-level communications routines and low-level communications routines. The categories are detailed below.

2.1 Control Routines

These routines are used to enable and disable high or low-level communications. They are usually invoked only once in a program. They include the following routines

close_com	Close communications, restores interrupt vectors
config_com	Overrides the default configuration file name (use prior to *open_com*)
open_com	Enables communications as specified in configuration file

2.2 High-level Communications Routines

These routines are used during high-level communications which involves robust message handling with error-free messages and message acknowledgement using stop-and-wait ARQ. The following routines are used in high-level communications:

flush_com	Resets a communication link after too many errors
get_com	Gets an available message from any link, checks for errors and terminal conditions (also provides the handshaking, so it must be called repeatedly)
look_com	Provides the station number given the high-level link station name
messtr	Provides the message type string given a message type number
ready_com	Checks to see if a link is ready for sending
send_com	Asynchronously sends a message to the selected destination
stnlstr	Provides the long station name given the station number
stnstr	Provides the short station name given the station number

2.3 Low-level Communications Routines

These routines are used during low-level communications which involve the sending and receiving of individual characters or character strings. There is no handshaking, error detection or translation involved. These routines are meant primarily for instrument control or to allow custom protocols to be implemented. They are necessary to allow use of serial ports serviced by the communication software but not used for high-level communications. Included are the following routines:

getc_low	Gets a character from a link
gets_low	Gets a terminated string from a link
look_low	Provides the station number given the low-level link station name
putc_low	Send a character to a link
puts_low	Send an unterminated string to a link

Note the *get_com*, while being a high-level routine, can be used in a strictly low-level system to detect the terminal condition of control-C/control-break being pressed. It has no other effect on low-level links.

3. Note on Program Fragments

With each routine description in the pages that follow, an example program fragment is included to illustrate the routine's use. It should be noted that despite appearances, these are not complete programs. The declarations necessary to understand the example are included at the beginning of the code. In many cases, opening and closing of services is omitted from the program fragments but must be included in a complete program.

close_com

Description: Closes all high and low-level communications including the restoration of all interrupt vectors for the serial ports, timers, control-C/control-break handlers and critical error handler.

Declaration: void close_com(void)

Parameters: **none**

Returns: **none**

Use: This routine must be called prior to program exit to restore the normal interrupt and critical error handlers for use with DOS. If it is not called, DOS will most probably hang up - there is also a chance that files could be corrupted. The programmer must ensure that this routine is called for normal exits, error exits and even for program aborts.

In the event of a critical error (such as floppy disk read) where Abort is chosen as a response, the critical error handler will automatically restore the vectors before returning control to DOS. This is because DOS does not return to the program when Abort is chosen.

Errors: **none**

Example:

```
int mlogger;                    // Logger station number
:
if ((mlocal=open_com()) == BAD_STATION) {
      printf("Error in starting comms in open_com.\n");
      // No close_com here because the open was unsuccessful
      exit(1);
}
if ((mlogger=look_com("data_logger")) == BAD_STATION) {
      printf("Cannot find link for data logger\n");
      close_com();
      exit(1);
}
while (send_com(mlogger,LOG,"This is a test message") != 0);
while (ready_com(mlogger) != 0);     // Wait for message to be sent
close_com();
exit(0);
```

Related Routines: *open_com*

config_com

Description: Overrides that default name (SERIAL.CFG) of the configuration file with a user specified name. Valid for high or low-level communications.

Declaration: void config_com(char *string)

Parameters: char *string   Pointer to the name of the configuration file (input)

Returns: **none**

Use: This routine is used to allow different configuration files to be used by different programs running in the same directory. By calling this routine prior to *open_com* a different configuration file or drive and path can be chosen. Without this routine, *open_com* looks for SERIAL.CFG in the default directory.

The name of the configuration file can be any DOS filename including file extension and, if desired, path and drive specifications. It is recommended, but not essential, that the extension ".CFG" be used for all such filenames. The filename must be a null-terminated string.

Errors: There is no return and therefore no error return. If the filename specified is not a valid filename, then the subsequent call to *open_com* will return BAD_STATION.

Example:

```
int mlocal;                            // Station number of local station
⋮
config_com("C:\EHF\NEWFILE.CFG");      // Use C:\EHF\NEWFILE.CFG instead of
                                       //  SERIAL.CFG in default directory
if ((mlocal=open_com()) == BAD_STATION) {
      printf("Error in starting comms in open_com.\n");
      exit(1);
}
⋮
```

Related Routines: *open_com*

flush_com

Description: Resets a link including the link consecutive error count and the total error count for all links. Used on high-level links but no effect on low-level links.

Declaration: int flush_com(int dest)

Parameters: int dest — Destination station number for the link obtained using *look_com* (input)

Returns: Integer, one of:

0	Link successfully reset
1	Bad station number

Use: This routine, not meant for general use, resets a link including: state, consecutive error count, receive & transmit buffers, receive & transmit ack flags and receive & transmit old message buffers. As well, the total error count for all links is reset.

This routine reinitializes a link and can be used to restart a link that failed because of too many errors. Generally, if a link receives too many errors, than the condition generating these errors (software at the far end faulty or not running, insufficient frequency of calls to *get_com*, poor choice of timeouts, control characters in message text, cables disconnected) must be corrected before using this routine. Because of this fact, it is unlikely that calling this routine, except under operator control, will provide any useful results.

This routine was developed because one cannot call *open_com* to restart communications without *close_com* which would disable all communications.

Errors: The destination station number is not valid (use *look_com* to get the valid number)

Example:

```
int mlogger;                    // Logger station number
⋮
if ((mlogger=look_com("data_logger")) == BAD_STATION) {
      printf("Cannot find link for data logger\n");
      exit(1);
}
if (flush_com(mlogger) != 0) {
      printf("Cannot reinitialize data logger link\n");
      exit(1);
}
⋮
```

Related Routines: none

get_com

Description: Gets any high-level message available, processes all incoming data and performs handshaking, checks for errors and terminal conditions. This routine must be called periodically for high-level communications to work.

Declaration: int get_com(int *ctype, int *cfrom, char *cdata)

Parameters:

int *ctype — Pointer to received message, error or termination type (output)
int *cfrom — Pointer to originating station (output)
char *cdata — Pointer to buffer, at least 200 characters long, contains the received message or the communications error message (output)

See table below for more details.

Returns: Integer, one of:

NO_MESSAGE	No messages available (any necessary processing occurred)
VALID_MSG	Valid receive messages returned in parameters
COMM_ERR	Communication error message returned in parameters
QUIT	Terminal condition occurred, no more communications and an orderly shut-down (*close_com*) should be done

Use: This routine must be called frequently during the execution of the user program to ensure that all high-level processing occurs (it is not compulsory for low-level communications but if called, it can report control-C and control-break key presses). The time between calls should be at no greater than 1/10th of the shortest timeout period (the default is timeout is 2 s). This routine should not be called until after *open_com*.

All of the high-level processing occurs in this routine: checksum processing and protocol handling as well as terminal condition detection. This routine is designed to be called repeatedly and does not take an excessive amount of processing time. If it is not called often enough, errors in handshaking occur usually noticed by messages or acks lost followed by duplicate messages or extra acks.

The following table summarizes the use of the parameters for the various return types:

Return Value	*ctype	*cfrom	*cdata
NO_MESSAGE	*unused*	*unused*	*unused*
VALID_MSG	Message Type	Originator	Message Data
COMM_ERR	Error Number	Originator	Error Text
QUIT	Termination Type	Originator (only for maximum consecutive errors)	*unused*

Message type is the type of message, as specified by the header and will be one of COMMAND, CONFIGURE, LOG, STATUS, POINT, MOD_POINT, TIME_STAMP

and ERROR. The text of the message type is available using the routine *messtr*.

The originator is the station number of the station at the far end of the link. The text name of this station is available using the routines *stnstr* or *stnlstr*.

The message data is a standard null-terminated string giving all the data portion of the message (the header is not included). For messages with no data (just header) this will be a null string.

Error number is the number of the error message. It is used internally and is not recommended for the user. Error text contains the textual error message including any parameters. It is a null-terminated string.

The termination type is one of:

TOTAL	Too many total errors occurred (sum of all errors on all links)
CONSEC	Too many consecutive errors on any link (originator specifies which link had too many errors)
BREAK	Control-C or control-break key occurred

Errors: This routine has no error returns itself, though a normal return can indicate a communications error. For high-level communications errors, see Communications Software User's Guide in Appendix A.

Example:

```
int mstat;                // Message status - valid, error or quit
int mtype;                // Message type number
int mfrom;                // Message from station number
char mdata[220];          // Message data
char string[220];         // String buffer used to name, type or error
:
mstat = get_com(&mtype,&mfrom,mdata);      // Check for message
if (mstat == VALID_MSG) {                  // Message available
      printf("    from %s ",stnstr(mfrom,string));
      printf("(%s):  \"%s\"\n",messtr(mtype,string),mdata);
} else if (mstat == COMM_ERR) {            // Communications error
      printf("-- Comm error with %s:  %s\n",stnlstr(mfrom,string),mdata);
} else if (mstat == QUIT) {                // End main program
      if (mtype == TOTAL) {                // Too many errors
            printf("Too many communication errors\n");
      } else if (mtype == CONSEC) {        // Too many in a row
            printf("Too many consecutive communications errors with %s\n",
                  stnlstr(mfrom,string));
      } else if (mtype == BREAK) {         // Control-C/Break
            printf("Break detected\n");
      }
      close_com();
      exit(1);
}
:
```

Related Routines: *send_com, getc_low, gets_low*

getc_low

Description: Gets a single character, if available, from a low-level link

Declaration: int getc_low(int dest)

Parameters: int dest — The sending station number at the other end of the link as obtained from *look_low* (input)

Returns: Integer containing the character received. If no characters are available, NO_DATA is returned. BAD_DEST is returned if the station number is not valid.

Use: This low-level link routine is the simplest way to get a character from the serial link. It checks to see if any characters are stored in the interrupt service routine's ring buffer and returns a character if available. No protocols are used nor do any translations occur.

Errors: The return BAD_DEST occurs when the station number is not a valid low-level link station. One must ensure the *look_low* routine is used to get the station number.

Example:

```
int c;              // Character received
int mmodem;         // Comstream Modem station number
:
if ((mmodem=look_low("comstream")) == BAD_STATION) {
        printf("Cannot find port for Comstream Modem.\n");
        close_com();
        exit(1);
}
c = getc_low(mmodem);
printf("The character received from the modem is %c\n",c);
:
```

Related Routines: *putc_low, gets_low, get_com, look_low*

gets_low

Description: Get a terminated string from a low-level link

Declaration: int gets_low(int dest, int term, char *string)

Parameters:

int dest	Sending station number as obtained from *look_low* (input)
int term	Terminating character for the string (input)
char *string	Pointer to buffer to receive the string (output)

Returns: Integer, one of:

ALL_OK	Valid string returned in buffer
NO_DATA	No data available
BAD_DEST	The station number is not valid

Use: This routine retrieves a string from the serial link specified. The characters are removed from the interrupt service routine's ring buffer and stored until the terminator is reached (while returning NO_DATA) and then the whole string, less terminating character, is returned. The received string is stored with a null terminator and is no longer than 200 characters.

If the terminator does not exist in the receive ring buffer, then *gets_low* will return NO_DATA. A later call to *gets_low* will can retrieve the data if the terminator is subsequently present in the ring buffer or the routine *getc_low* can be used to get at the characters one at a time.

Errors: The return BAD_DEST occurs when the station number is not a valid low-level link station. One must ensure the *look_low* routine is used to get the station number.

Example:

```
char inline[220]; // Input line buffer
int mmodem;       // Comstream Modem station number
⋮
if ((mmodem=look_low("comstream")) == BAD_STATION) {
      printf("Cannot find port for Comstream Modem.\n");
      close_com();
      exit(1);
}
c = gets_low(mmodem,'\n',inline);
printf("The line received from the modem is %s\n",inline);
⋮
```

Related Routines: *puts_low, getc_low, get_com, look_low*

look_com

Description: Provides the station number given the long station name for a high-level link.

Declaration: int look_com(char *stn)

Parameters: char *stn — Pointer to station name (input)

Returns: Integer station number for the station name. If the station name is not recognized, BAD_STATION is returned.

Use: This routine is used to get the station number for high-level communications prior to using the routine *send_com*. It first determines if the station name is one of the valid names: DATA_LOGGER, BEACON_MON, BURST_DEMOD, TX_PROC, EPHEM_PROC, SYNC_PROC, CRC_ANTENNA or T85_ANTENNA. Then it checks all the links defined by the configuration file and determines if the station name occurs in one of the link definitions (in other words that station is connected to this computer). If all checks out, then the station number is returned.

The station name string is a null-terminated string where case is unimportant. It must be free of blanks and control characters.

Errors: A return value of BAD_STATION can be caused by:
- a spelling error in the long station name
- blanks or control characters in the long station name
- giving the station name of a low-level link (use *look_low* instead)
- an attempt to use the short station name (4 characters) instead of the long one
- the configuration file does not define a link to the given station name

Example:

```
int mlogger;                    // Logger station number
⋮
if ((mlogger=look_com("data_logger")) == BAD_STATION) {
      printf("Cannot find link for data logger\n");
      exit(1);
}
⋮
```

Related Routines: *look_low*, *send_com*, *stnlstr*, *stnstr*

look_low

Description: Determines the station number given a low-level link station name

Declaration: int look_low(char *stn)

Parameters: char *stn   Pointer to low-level station name string (input)

Returns: Integer station number associated with the station name. If the station name is not recognized, BAD_STATION is returned.

Use: This routine is used to get the station number for low-level communications prior to using any of the following routines: *getc_low*, *putc_low*, *gets_low* or *puts_low*. It checks the name against all of the names used in the low-level declarations in the configuration file.

The station name string is a null-terminated string where case is unimportant. It must be free of blanks and control characters.

Errors: A return values of BAD_STATION can be caused by:
- a spelling error in the station name
- blanks or control characters in the station name
- giving a station name for a high-level link (use *look_com* instead)
- the configuration file does not define a link to the given station name

Example:

```
int mmodem;         // Comstream Modem station number
⋮
if ((mmodem=look_low("comstream")) == BAD_STATION) {
      printf("Cannot find port for Comstream Modem.\n");
      close_com();
      exit(1);
}
⋮
```

Related Routines: *look_com*, *getc_low*, *putc_low*, *gets_low*, *puts_low*

messtr

Description: Provides a message type string for a given message type

Declaration: char *messtr(int n, char *string)

Parameters:

int n	Message type number obtained from *get_com* (input)
char *string	Pointer to the buffer to contain the message type string (output)

Returns: Pointer to the buffer that contains the message type string. This pointer is identical to the parameter. There is no error return.

Use: When provided with a message type number, as returned by *get_com*, this routine returns the message type as a fixed-length null-terminated string. The string is entirely in lower case with training blanks to make 6 characters.

Note that there is no validation of the message type number, so a bad message type can cause unknown results.

Errors: None, but the use of an invalid message type number can cause unpredictable results.

Example:

```
int mstat;                  // Message status - valid, error or quit
int mtype;                  // Message type number
int mfrom;                  // Message from station number
char mdata[220];            // Message data
char string[220];           // String buffer used for name or type
:
mstat = get_com(&mtype,&mfrom,mdata);     // Check for message
if (mstat == VALID_MSG) {                 // Message available
      printf("    from %s ",stnstr(mfrom,string));
      printf("(%s):  \"%s\"\n",messtr(mtype,string),mdata);
}
:
```

Related Routines: *get_com, stnstr, stnlstr*

open_com

Description: Opens all high and low-level communications including set-up for control-C and critical error trappings. Reads in all the configuration information from the configuration file.

Declaration: int open_com(void)

Parameters: **none**

Returns: Integer station number of the local station. If an error occurred, BAD_STATION is returned.

Use: This routine should be called only once prior to any communications, high or low-level. The ports cannot be reconfigured by a later call - in fact a second call will always result in an error.

The routine sets up the serial ports, timers, enables serial port interrupts and redirects the control-C/control-break and critical error handlers. The interrupts and handlers must be restored by using *close_com* prior to ending the program or DOS will likely hang up.

This routine reads in the configuration file to determine the settings for the serial ports. This file defaults to SERIAL.CFG in the default directory but any name specified by a prior call to *config_com* can be used.

When successfully invoked, this routine prints out a two line header that indicates the board type used, the name of the configuration file and the software versions of COM.H, COM.C and SERIAL.ASM

Errors: A return value of BAD_STATION can be caused by
- the configuration file can not be opened (doesn't exist or is already in use)
- an error in occurred in the configuration file (supplementary message will be displayed)

Example:

```
int mlocal;              // Station number of local station
char string[220];        // String buffer used to hold station name
⋮
if ((mlocal=open_com()) == BAD_STATION) {
      printf("Error in starting comms in open_com.\n");
      exit(1);
}
printf("Local station is %s\n",stnlstr(mlocal,string));
⋮
```

Related Routines: *close_com, config_com*

putc_low

Description: Send a character out a low-level link

Declaration: int putc_low(int dest, int c)

Parameters:

int dest	Receiving station number as obtained from *look_low* (input)
int c	Character to be sent (input)

Returns: Integer, one of:

ALL_OK	The character was successfully passed to the serial port interrupt subroutine to be transmitted on the next interrupt
BAD_DEST	The receiving station number is not valid

Use: This routine is the simplest way to send a character out a serial link. It loads the character into the serial port interrupt service routine's ring buffer to be sent out on the appropriate interrupt. No protocols or translations are used.

Note that it is possible to put characters into the ring buffer faster than the service routine can service them. In general, no more than 500 characters should be put into the ring buffer without ensuring that they have been sent. This could be by using some special protocol (such as a response to a command), using a time delay (baud rate/10 gives the number of characters per second) or by examining echoed characters.

Errors: The return BAD_DEST occurs when the station number is not a valid low-level link station. One must ensure the *look_low* routine is used to get the station number.

Example:

```
int c;              // Character to be sent
int mmodem;         // Comstream Modem station number
⋮
if ((mmodem=look_low("comstream")) == BAD_STATION) {
      printf("Cannot find port for Comstream Modem.\n");
      close_com();
      exit(1);
}
printf("Enter character to be sent to the modem?");
c = getch();
putc_low(mmodem,c);
⋮
```

Related Routines: *getc_low, puts_low, send_com, look_low*

puts_low

Description: Sends an unterminated string out a low-level link

Declaration: int puts_low(int dest, char *string)

Parameters:

int dest	Receiving station number as obtained from *look_low* (input)
char *string	Pointer to string to be sent (input)

Returns: Integer, one of:

ALL_OK	The string was successfully passed to the serial port interrupt subroutine to be transmitted in sequence
BAD_DEST	The receiving station number is not valid

Use: This routine takes a null-terminated string and sends it out the low-level link less the null termination. If terminations are required as part of the protocol (such as a linefeed at the end of the line) then the terminating character must be included in the string. The string is loaded into the serial port interrupt service routine's ring buffer to be sent out on the appropriate interrupts. No protocols or translations are used.

Because this routine take a null-terminated string as input, it cannot be used to send a null. If it is desired to send a null within or at the end of the string, a separate call to *putc_low* must be made to send the null.

As with the routine *putc_low*, it is possible to put characters into the ring buffer faster than the service routine can service them. In general, no more than 500 characters should be put into the ring buffer without ensuring that they have been sent out. This could be by using some special protocol (such as a response to a command), using a time delay (baud rate/10 gives the number of characters per second) or by examining echoed characters.

Errors: The return BAD_DEST occurs when the station number is not a valid low-level link station. One must ensure that the *look_low* routine is used to get the station number.

Example:

```
char s[220];        // String to be sent to the modem
int mmodem;         // Comstream Modem station number
:
if ((mmodem=look_low("comstream")) == BAD_STATION) {
      printf("Cannot find port for Comstream Modem.\n");
      close_com();
      exit(1);
}
printf("Enter string to be sent to the modem?");
scanf("%s",s);
puts_low(mmodem,s);
:
```

Related Routines: *gets_low, putc_low, send_com, look_low*

ready_com

Description: Checks to see if a high-level link is ready for sending.

Declaration: int ready_com(int dest)

Parameters: int dest — Destination station number for the link obtained using *look_com* (input)

Returns: Integer, one of:

0	Link is ready for sending
1	Link not ready because the link is still transmitting or bad station number

Use: This routine checks to see if the transmit buffer for the link is available. Normally, this routine is passed a legal destination station number so the not-ready return means that the previous message is still being transmitted or is waiting for an ack. Because of a possible requirement for retransmission, the buffer must hold any outgoing message until the ack is received.

This routine is most often used prior to program termination to ensure that all outstanding messages have been sent and acknowledged prior to exiting. Similar return values can be obtained from the routine *send_com* if one is only waiting to transmit the next message.

Errors: A return of link-not-ready can occur if one of the following:
- the link is not ready because the preceding message has not yet completed the transmission or handshaking
- the destination station number is not valid (use *look_com* to get the valid number)

Example:

```
int mlogger;                          // Logger station number
:
if ((mlogger=look_com("data_logger")) == BAD_STATION) {
      printf("Cannot find link for data logger\n");
      exit(1);
}
while (send_com(mlogger,LOG,"This is a test message") != 0)
      checkmsg();                     // Check break and get_com
while (ready_com(mlogger) != 0)      // Wait for message to be sent
      checkmsg();                     // Check break and get_com
:                                     //  (checkmsg is documented on page 38)
```

Related Routines: *send_com*, *look_com*

send_com

Description: Asynchronously sends one high-level message to the selected destination if it is ready. It formats the message and ensures reliable transfer with stop-and-wait ARQ.

Declaration: int send_com(int dest, int mtype, char *string)

Parameters:

int dest	The destination station number as obtained from *look_com* (input)
int mtype	The message type number (input)
char *string	Pointer to the message text, null string for header only (input)

Returns: Integer, one of:

0	Normal return, no error
1	Message not sent because of link not ready or illegal destination number

Use: This routine formats the message by putting originator, destination, message type and checksum in the header and adding on the message text and delimiters. It then places the outgoing message in the buffer, begins to send it and returns. The remaining transmissions and handshaking take place under interrupt control and through repeated calls to *get_com* to process the handshaking.

Normally, this routine is passed legal destination numbers, so the message-not-sent return value is indicative of the link not ready. This is because either the preceding message has not yet finished transmission or the ack is still outstanding. Because of a possible requirement for retransmission, the buffer must hold any outgoing message until the ack is received. The message-not-sent return value of this routine can be used to wait for the link to be ready, or *ready_com* can be used to simply check for the ready state.

The message type must be one of: COMMAND, CONFIGURE, LOG, STATUS, POINT, MOD_POINT, TIME_STAMP or ERROR. The message text must be a null-terminated string no longer than 199 characters but may be a null string. **The message text must not contain any control characters, especially not linefeeds or carriage returns which are used as message delimiters in high-level protocol.**

Errors: A return of message-not-sent can occur if one of the following:
- the link is not ready because the preceding message has not yet completed the transmission or handshaking (*ready_com* can be used to check readiness of link)
- the destination station number is not valid (use *look_com* to get the valid number)

Example:

```
int mlogger;                    // Logger station number
:
if ((mlogger=look_com("data_logger")) == BAD_STATION) {
      printf("Cannot find link for data logger\n");
      exit(1);
}
while (send_com(mlogger,LOG,"This is a test message") != 0);
:
```

Related Routines: *get_com, putc_low, puts_low, look_com, ready_com*

stnlstr

Description: Provides the long station name for a given high-level station number

Declaration: char *stnlstr(int n, char *string)

Parameters:

int n	Station number for the high-level link (input)
char *string	Pointer to the buffer to contain the long station name (output)

Returns: Pointer to the buffer that contains the long station name. This pointer is identical to the parameter. There is no error return.

Use: When provided with a high-level station number, as returned by *look_com*, this routine returns the long (variable length) station name in a null-terminated string. This name is entirely in lower case. This routine is often used when outputting the details of a received message from *get_com*. For a short (4 character) fixed-length name, use *stnstr*.

This routine only works for high-level link names. Low-level link names must already be known within the program so there is no equivalent routine for low-level link names.

Note that there is no validation of the station number, so a bad station number can cause unknown results.

Errors: None, but the use of an invalid station number can cause unpredictable results.

Example:

```
int mlocal;                 // Station number of local station
char string[220];           // String buffer used to hold long station name
⋮
if ((mlocal=open_com()) == BAD_STATION) {
      printf("Error in starting comms in open_com.\n");
      exit(1);
}
printf("Long station is %s\n",stnlstr(mlocal,string));
⋮
```

Related Routines: *stnstr*, *messtr*, *look_com*

stnstr

Description: Provides the short station name given the station number for a high-level link

Declaration: char *stnstr(int n, char *string)

Parameters:

int n	Station number for the high-level link (input)
char *string	Pointer to buffer to contain the short station name (output)

Returns: Pointer to the buffer that contains the short station name. This pointer is identical to the parameter. There is no error return.

Use: When provided with a high-level station number, as returned by *look_com*, this routine returns the short (4 character) station name in a null-terminated string. This name is entirely in lower case and relatively cryptic - its primary use is in message headers of the high-level protocol. For a more understandable name use *stnlstr*.

This routine only works for high-level link names. Low-level link names must already be known within the program so there is no equivalent routine for low-level link names.

Note that there is no validation of the station number, so a bad station number can cause unknown results.

Errors: None, but the use of an invalid station number can cause unpredictable results.

Example:

```
int mlocal;                 // Station number of local station
char string[10];            // String buffer used to hold short station name
⋮
if ((mlocal=open_com()) == BAD_STATION) {
      printf("Error in starting comms in open_com.\n");
      exit(1);
}
printf("Short station is %s\n",stnstr(mlocal,string));
⋮
```

Related Routines: *stnlstr*, *messtr*, *look_com*

Appendix C

Real-time Software Programmer's Reference

1. Introduction

This appendix provides all the use and interface details for the real-time routines used by the communications software. These routines provide control of the hardware that is not easily done in a higher level language. Although they were designed to support the communications software, they are also of use for other programs to provide interrupt driven serial communications, timer support and control over user initiated aborts through control-C/control break and critical error trapping.

Since these are assembly language routines, there is no header file associated with their declarations. To use these routines in a C program, function prototypes must be used based on the declaration given for the specific routine in the following pages.

The routines, in the file SERIAL.ASM, were assembled using Microsoft Assembler 5.10 under DOS 5.0 and are based on the small memory model. To use the large memory model, they must be reassembled with different stack parameter offsets. See the "Memory Model Size" section in the declaration area of the SERIAL.ASM program listing.

These routines are grouped into four categories: serial port, timer support, control-C/control-break trapping and critical error trapping. These four categories are independent and stand-alone with the exception of the critical error handler, which, upon detection of Abort, shuts down the other three services. Each category is detailed below.

2. Serial Port Routines

These routines allow interrupt driven serial port communications. Unlike the DOS and BIOS calls which only provide polled communications, these routines receive and transmit data on an interrupt basis so they do not tie up the processor when waiting for data. Simple character read and write services are provided along with opening, closing and configuring the serial ports. A composite status (an ORing operation for the errors from all ports, ex: if the transmit buffer overflow bit is set then at least one port had a transmit buffer overflow) is also available. The software supports up to ten ports and can easily be extended with recompilation. The serial port routines are:

close_ser	Closes a serial port and restores interrupts
open_ser	Opens a serial port
read_ser	Reads a character from a serial port
set_ser	Sets the baud rate, bits, stop bits and parity of a serial port
stat_ser	Returns the composite status of all serial ports
ver_ser	Return the version number string for the serial port software
write_ser	Sends a character to a serial port

3. Timer Routines

The timer routines provide eleven countdown timers used mostly by the communications software for measuring timeouts. They can also be used for general purpose delays of up to 30 minutes. Routines are provided to open, close, set and check the remaining count for a timer. Each timer counts down to 0 and remains there. The timer routines are:

chk_time	Returns remaining number of ticks for a countdown timer
close_time	Closes all countdown timers and restore interrupts
open_time	Initializes and enables all 11 timers
set_time	Sets the tick count for a countdown timer

4. Control-C/Control-Break Detection Routines

These routines allow the trapping of control-C and control-break. If a user presses either of the key combinations, DOS normally aborts the program and returns to the prompt. Software using interrupts must restore them prior to exiting, so trapping control-C/control-break keypresses allow the programmer to do a clean exit rather than the abort forced by DOS. Routines are provided to open, close and to check for the control-C/control-break keypresses. The control-C/control-break detection routines include:

close_break	Restores DOS control-C/control-break handler
open_break	Enables trapping of control-C/control-break
press_break	Checks to see if control-C/control-break was pressed

5. Critical Error Handler Routines

These routines allow critical errors to be trapped and, upon abort, the interrupts restored prior to control being returned to DOS. Critical errors usually deal with printers or disk drives (for example "Drive Not Ready" when there is no floppy in the drive). Without trapping critical errors, the user could abort the program without allowing the interrupts to be restored.

The handler installed by these routines, upon abort, closes serial, timer and control-C/control-break. The close routines associated with the services are robust and work even if the associated service has not been enabled. It is this hard-coding of closures that makes these routines specific to the rest of the SERIAL.ASM routines. If other interrupts are to be closed, this must be added to the code of the critical error handler.

The critical error handler routines are:

close_crit	Restores DOS critical error handler
open_crit	Enables trapping of critical errors

chk_time

Description: Returns number of ticks remaining in countdown timer

Declaration: int chk_time(int timer)

Parameters: int timer — Countdown timer to be set with a valid range of 0-10

Returns: Integer, one of:

1 to 32767	Number of ticks remaining in countdown
0	Timer countdown complete
-1	Timer number out of range

Use: This routine is used to check for completion of the countdown timer. The user software should be checking until the value returned is 0. Note that once the timer reaches 0, it remains there so this routine only guarantees that the timeout period has been exceeded. The amount that it has been exceeded depends on the frequency of calls to this routine.

Errors: Timer numbers must be within the range 0-10.

Example:

```
⋮
open_time();
set_time(6,50);                 // Set timer 6 for 50 ticks (3 s)
while (chk_time(6) != 0);       // Wait for 3 s
close_time();
⋮
```

Related Routines: *set_time*

close_break

Description: Restores default control-C/control-break handler

Declaration: void close_break(void)

Parameters: **none**

Returns: **none**

Use: This routine disables control-C/control-break trapping and restores the DOS default handler. The routine first checks to see if trapping was enabled (by an earlier call to *open_break*). In the case that trapping was not previously enabled this routine just exits.

Errors: **none**

Example:

```
⋮
open_break();
while (press_break() == 0);    // Wait for break to be pressed
close_break();
⋮
```

Related Routines: *open_break, press_break*

close_crit

Description: Restores system critical error handler

Declaration: void close_crit(void)

Parameters: **none**

Returns: **none**

Use: This routine disables critical error trapping and restores the DOS critical error handler. It is used just prior to exiting to DOS by a program that uses interrupts. This routine must be called last, after all other interrupts have been restored by closing.

Errors: **none**

Example:

```
⋮
      // Critical error use the normal DOS handler
open_crit();
      // Critical errors are now trapped
⋮
close_crit();
      // Critical errors use the normal DOS handler
⋮
```

Related Routines: *open_crit*

close_ser

Description: Closes serial port by restoring interrupts

Declaration: int close_ser(int port)

Parameters: int port — Serial port to be disabled with a valid range of 1-10 for COM1-COM10

Returns: Integer, one of:

0	Successfully closed
1	Port could not be closed

Use: This routine should be called prior to exiting for each port that was used. Once called, the interrupts for that port are disabled and if no other active ports are using that vector, the vector is restored.

After all ports are closed (therefore all interrupts have been restored) then *close_crit* should be called to restore the default critical error handler.

With a valid port number, one is guaranteed that the port is closed after the call - either it is closed with the call or was already closed. So if it is not known which ports are active, then the programmer can close all ports and disregard the return value.

Errors: The port cannot be closed if:
- the port number is out of the range 1-10 for COM1-COM10
- the port is already been closed or was never opened

Example:

```
int i;                                  // Integer index
:
for (i=1;i<=10;i++)
            close_ser(i);               // Close all ten ports
close_crit();                           // Restore default critical error handler
exit(0);
```

Related Routines: *open_ser*

close_time

Description: Disables all countdown timers and restores interrupts

Declaration: void close_time(void)

Parameters: none

Returns: none

Use: Turns off the countdown timers and restores the default interrupt service routines. This routine first checks to see if the timers were previously enabled. If not, then no action is taken.

Errors: none

Example:

```
:
open_time();
set_time(6,50);              // Set timer 6 for 50 ticks (3 s)
while (chk_time(6) != 0);    // Wait for 3 s
close_time();
:
```

Related Routines: *open_time*

open_break

Description: Setup the control-C/control-break handler

Declaration: void open_break(void)

Parameters: **none**

Returns: **none**

Use: This routines allows the trapping of control-C and control-break. They must be trapped to ensure that a program using interrupts can have an orderly exit if the user decides to abort.

If the routine has already been called (and not closed), this and subsequent calls to open the control-C/control-break handler are ignored.

Errors: **none**

Example:

```
⋮
open_break();
while (press_break() == 0);    // Wait for break to be pressed
close_break();
⋮
```

Related Routines: *close_break, press_break*

open_crit

Description: Enables critical errors to be trapped and clean exit upon Abort

Declaration: void open_crit(void)

Parameters: **none**

Returns: **none**

Use: This routine allows critical errors to be trapped to a handler that restores the default interrupts prior to allowing an Abort exit. Because the Abort exit to a critical error does not return to the user software, any programs that use interrupts must restore them at an Abort exit to a critical error prior to returning to DOS with the Abort return.

This routine must be invoked prior to any routines using interrupts.

A critical error normally includes information relating to the cause of the error (drive letter, type of problem). For simplicity, the critical error handler used here only reports a critical error without specifying the source of the problem.

The critical error handler has a hard coded calls to *close_break*, *close_time* and *close_ser*. All these routines are programmed so that they can be called without crashing even if the corresponding open has not been done. If any other interrupts are used, then a closing call must be added to this routine. Thus this routine is specific to the serial communications software.

Errors: **none**

Example:

```
:
      // Critical error use the normal DOS handler
open_crit();
      // Critical errors are now trapped
:
close_crit();
      // Critical errors use the normal DOS handler
:
```

Related Routines: *close_crit*

open_ser

Description: Sets up interrupts and initialize to allow communications on the specified port

Declaration: int open_ser(int port, int type)

Parameters:

int port	Serial port to be enabled with a valid range of 1-10 for COM1-COM10	
int type	Type of serial board used	
	0	Standard COM1-COM4 addresses
	1	Digiboard addresses COM3-COM10 (standard COM1 & COM2)

Returns: Integer, one of:

0	Successfully opened
1	Port could not be opened

Use: This routine must be called once for every port to be used. Once called, the serial interrupt is redirected (if not already done) and then the UART is initialized. This routine initializes the UART, but does not set the baud rate and associated parameters - use *set_ser* to do this.

The Digiboard PC/8 when installed uses different addresses for COM3 and COM4 and allows the use of COM5 to COM10 (not defined for a PC). The serial board type parameter allows the software to work on computer with normal PC serial ports or with the Digiboard PC/8 installed. If multiple ports (and therefore multiple opens) are used, the type of board must be the same for all calls.

The critical error handler, set up by *open_crit*, should be invoked prior to this routine. If it is not used and a user selects Abort in response to a critical error (such as floppy drive not ready) then DOS will likely hang because the interrupt vectors will not have been restored.

Errors: The port can not be opened if:
- the port number is out of the range 1-10 for ports COM1-COM10
- the port is already open

Example:

```
:
open_crit();       // Ensure critical error handler active
if (open_ser(3,1) != 0) {          // COM3 using Digiboard
      printf("Cannot open COM3\n");
      close_crit();
      exit(1);
}
set_ser(3,0xE3);   // 9600 baud, no parity, 1 stop bit, 8 bits/character
:
```

Related Routines: *close_ser*, *set_ser*

open_time

Description: Initializes and enables all countdown timers

Declaration: void open_time(void)

Parameters: **none**

Returns: **none**

Use: This routine sets up the interrupts necessary to enable the 11 count-down timers. These timers were designed to be used as timeout timers.

If the routine has already been called (and not closed), this and subsequent calls to open the timers are ignored.

Errors: **none**

Example:

```
:
open_time();
set_time(6,50);                // Set timer 6 for 50 ticks (3 s)
while (chk_time(6) != 0);      // Wait for 3 s
close_time();
:
```

Related Routines: *close_time*

press_break

Description: Checks to see if control-C/control-break was pressed since last invocation

Declaration: int press_break(void)

Parameters: **none**

Returns: Integer, one of:

0	No control-C/control-break pressed
27	Control-break pressed since last call
35	Control-C pressed since last call

Use: This routine checks to see if either control-C or control-break have been pressed since the last call (or since *open_break*). As long as the return value is 0, no keys have been pressed requesting a program abort.

This routine traps all occurrences of control-C or control-break, so upon detection, the programmer must implement a clean-up routine of the various interrupts.

Note that there is only one flag for control-C and control-break. If both are pressed, only the last one pressed will be returned.

Errors: **none**

Example:

```
:
open_break();
while (press_break() == 0);    // Wait for break to be pressed
close_break();
:
```

Related Routines: *open_break*, *close_break*

read_ser

Description: Gets a character from a serial port

Declaration: int read_ser(int port)

Parameters: int port — Serial port to be read with a valid range of 1-10 for COM1-COM10

Returns: Integer value:

0 to 255	Character received
-1	No character available
-2	Port number is out of range

Use: This routine when called checks the receive ring buffer of the appropriate port for a character, and if one is available returns it. Otherwise, the routine returns with a no character available. This means that the routine can be called frequently without forcing the software to wait for the next character.

Errors: The port number is out of range if it is not within the range of 1-10 for COM1-COM10. A return of no character available means that no new character has been received yet for the serial port.

Example:

```
int c:                              // Character received
:
write_ser(1,'C');                   // Send "CF?" out port
write_ser(1,'F');
write_ser(1,'?');
while ((c=read_ser(1)) < 0);        // Wait for character
printf("Response is %c\n",c);
:
```

Related Routines: *write_ser*

set_ser

Description: Sets the serial port parameters: baud rate, number of bits/character, number of stop bits and parity.

Declaration: int set_ser(int port, int parm)

Parameters:

int port	serial port to be set with a valid range of 1-10 for COM1-COM10
int parm	serial port parameter (see table below)

Returns: Integer, one of:

0	Parameters successfully set
1	Port number out of range

Use: This routine sets the communication parameters for the serial port. The four parameters are fully specified in the low 8-bits of the integer parameter. This routine should be called immediately after *open_ser* and prior to any communications. This routine can be called again to later change the communication parameters.

The table below gives the values used to specify the four serial port parameters. One value must be selected for each parameter and summed to get the composite parameter.

Bits per Character		Stop Bits		Parity		Baud Rate	
5 bits	0x00	1 bit	0x00	None	0x00	110	0x00
6 bits	0x01	2 bits	0x04	Odd	0x08	150	0x20
7 bits	0x02			Even	0x18	300	0x40
8 bits	0x03					600	0x60
						1200	0x80
						2400	0xA0
						4800	0xC0
						9600	0xE0

Errors: Port number out of range occurs if the port number is not one of 1-10 for COM1-COM10

Example:

```
:
open_crit();       // Ensure critical error handler active
if (open_ser(3,1) != 0) {          // COM3 using Digiboard
      printf("Cannot open COM3\n");
      close_crit();
      exit(1);
}
set_ser(3,0xE3);   // 9600 baud, no parity, 1 stop bit, 8 bits/character
:
```

Related Routines: *open_ser*

set_time

Description: Sets a specific countdown timer to a tick count

Declaration: int set_time(int timer, int tick)

Parameters:

int timer	Countdown timer to be set with a valid range of 0-10
int tick	Number of ticks (16.7 ticks per second)

Returns: Integer, one of:

0	Timer successfully set
1	Timer number out of range

Use: This routine sets a countdown timer to a specific number of ticks. The counter will be decremented at each timer interrupt until it reaches 0 where it will remain (until set again). There are 16.7 ticks per second, so using these timers, with a positive integer tick count, the longest timeout period is 1962 s or almost 33 minutes. Negative values will provide unpredictable results and should not be used.

Note that the asynchronous nature of the timer setting and decrementing allow an ambiguity of just less than one tick (60 ms). Therefore, the minimum setting should be a value of 2 to ensure the period is at least one tick long. The smallest period is then 60-120 ms.

Errors: Timer numbers must be within the range 0-10.

Example:

```
:
open_time();
set_time(6,50);                 // Set timer 6 for 50 ticks (3 s)
while (chk_time(6) != 0);       // Wait for 3 s
close_time();
:
```

Related Routines: *chk_time*

stat_ser

Description: Provides a composite status of the ports

Declaration: int stat_ser(void)

Parameters: **none**

Returns: Integer status

0x01	Interrupt service routine invoked but no active port caused interrupt
0x02	Handshake line change caused interrupt, but it was supposed to be disabled
0x04	Serial line break or UART error
0x08	Receive ring buffer overflow
0x10	Transmit ring buffer overflow
0x20	Transmit ring buffer not empty

Use: This routine returns a composite status, with error conditions latched, of all of the active ports. The status is cleared after each call, so the bits indicate that at least one of the error events occurred since startup or the last call to this routine. The Transmit buffer not empty bit is not latched, it is simply the state of the transmit ring buffer at the time of the call. This bit can be used to wait for all data to be transmitted.

If too much data is sent using *write_ser* and there is insufficient time to send it, then the Transmit ring buffer overflow will be set. On the other hand, if lots of data is being received and no calls to *read_ser* are made, then eventually the Receive ring buffer overflow will be set.

If a serial line break (long period of space) occurred or there were asynchronous framing errors (such as no stop bit) then the Serial link break or UART error bit will be set.

The bad interrupts bits will be set only if there are other programs (such as TSRs) attempting to use the serial ports. This should not occur in normal operation.

Errors: **none**

Example:

```
:
write_ser(1,'H');                    // Send out "Hi\n"
write_ser(1,'I');
write_ser(1,'\n');
while ((stat_ser() & 0x20) != 0);    // Wait for buffer empty
:
```

Related Routines: **none**

ver_ser

Description: Returns the version number string of the serial port software

Declaration: char *ver_str(void)

Parameters: none

Returns: Pointer to character string with the serial port software version number. The version number starts with a "V", followed by a date and then a decimal version. (ex: V02Jun93.01 means that it was the first version created on June 2, 1993)

Use: This routine returns the version number string to allow user programs to know which version of the software has been linked. It is used by the communication software during opening to display all of the relevant software versions. Any modifications to the file SERIAL.ASM will result in an updated version number.

Errors: none

Example:

```
char *strpnt;
:
strpnt = ver_str();                            // Get the version number
printf("The version number of SERIAL.ASM is %s\n",strpnt);
:
```

Related Routines: none

write_ser

Description: Sends a character to a serial port

Declaration: int write_ser(int port, int ich);

Parameters:

int port	Serial port, valid range 1-10 for COM1-COM10, to which the character is to be sent
int ich	Character to be sent to the port with a valid range of 0-255

Returns: Integer, one of:

0	Character successfully send
1	Port number out of range

Use: This routine puts one character into the transmit ring buffer of the specified port. If the transmit ring buffer is full, the character is discarded and the Transmit ring buffer overflow bit is set (use *stat_ser* to check this bit).

Note that it is possible to put characters into the ring buffer faster than the service routine can service them. In general, no more than 500 characters should be put into the ring buffer without ensuring that they have been sent. This could be by using some special protocol (such as response to a command), using a time delay (baud rate/10 gives the number of characters per second), by examining echoed characters or by checking the composite Transmit ring buffer not empty bit available from *stat_ser*.

Errors: Port number out of range occurs if the port number is not one of 1-10 for COM1-COM10

Example:

```
:
write_ser(1,'H');                   // Send out "Hi\n"
write_ser(1,'I');
write_ser(1,'\n');
while ((stat_ser() & 0x20) != 0);   // Wait for buffer empty
:
```

Related Routines: *read_ser*, *stat_ser*

Appendix D

Communications Software Listing

1. Introduction

In this appendix, the two files, header file COM.H and the file COM.C, used for the communications software are listed. This appendix does not cover the assembly language routines which are given in the Appendix E - Real-time Software Listing.

2. Header File COM.H

```
#define HEAD_VERSION    "V02Jun93.01"

/*
 Station name definitions
------------------------- */
#define BAD_STATION     -1      /* Station name or number not valid     */
#define UNKNOWN_ID      0       /* Station name garbled or not sent     */
#define DATA_LOGGER     1       /* Data Logger & Experiment Controller  */
#define BEACON_MON      2       /* Beacon & Reference Monitor           */
#define BURST_DEMOD     3       /* Burst DPSK Demodulator Host          */
#define TX_PROC         4       /* CRC Transmit Processor               */
#define EPHEM_PROC      5       /* Ephemeris Processor                  */
#define SYNC_PROC       6       /* Synchronization Processor            */
#define CRC_ANTENNA     7       /* CRC Antenna Controller Host          */
#define T85_ANTENNA     8       /* T85 Antenna Controller Host          */

#define NSTATION        9       /* Number of valid stations             */
#define LENSTN          4       /* Length of station name field         */
#define LOW_BASE        20      /* Base number used for low-level ports */

#define SNAMES          "unkn","dlog","beac","bdem","txpr","ephm","sync","crca","t85a"
#define LNAMES          "unknown","data_logger","beacon_mon","burst_demod","tx_proc",\
                        "ephem_proc","sync_proc","crc_antenna","t85_antenna"

/*
 Receiver status definitions
----------------------------- */
#define NO_MESSAGE      0       /* No message ready received            */
#define VALID_MSG       1       /* Valid message received               */
#define COMM_ERR        2       /* Communications error occurred        */
#define QUIT            3       /* Exit program requested               */

/*
 Message type definitions
------------------------- */
#define BAD_MESSAGE     -1      /* Message is invalid                   */
#define ACK             0       /* Ack message                          */
#define NAK             1       /* Nak message                          */
#define COMMAND         2       /* Command message                      */
#define CONFIGURE       3       /* Configuration message                */
#define LOG             4       /* Log message                          */
#define STATUS          5       /* Status message                       */
#define POINT           6       /* Initial pointing information         */
#define MOD_POINT       7       /* Modified pointing information        */
#define TIME_STAMP      8       /* Time stamp                           */
#define ERROR           9       /* Error condition message              */

#define NMESSAGE        10      /* Number of message types              */
#define LENMSG          6       /* Length of message type field         */
```

```
#define MNAMES          "ack   ","nak   ","comd  ","config","log   ",\
                        "status","point ","modpnt","time  ","error "


/*
 Error check definitions
------------------------- */
#define NO_ERROR        0       /* No error occurred                          */
#define TOTAL           1       /* Too many total errors occurred             */
#define CONSEC          2       /* Too many consecutive errors on 1 port      */
#define BREAK           3       /* Control-Break or Control-C occurred        */

/*
 Low-level "get" return definitions
------------------------------------ */
#define BAD_DEST        -2      /* Destination number is invalid        */
#define NO_DATA         -1      /* No data is available                 */
#define ALL_OK          0       /* Normal return                        */

/*
 High-level communications routines
------------------------------------ */
/*      open_com        opens all high and low level communications     */
/*      get_com         gets one message, if available, returns status  */
/*      send_com        sends one message                               */
/*      look_com        determines port number given station name       */
/*      ready_com       checks to see if port is ready to send message  */
/*      config_com      overrides default SERIAL.CFG name               */
/*      flush_com       resets errors on a channel                      */
/*      close_com       closes all high and low level communications    */
int open_com(void);
int get_com(int *ctype, int *cfrom, char *cdata);
int send_com(int dest,int mtype,char *string);
int look_com(char *stn);
int ready_com(int dest);
void config_com(char *string);
int flush_com(int dest);
void close_com(void);

/*
 Low-level communications routines
----------------------------------- */
/*  These routines need high-level "open_com" and "close_com" before use */
/*      getc_low        gets one character                               */
/*      gets_low        gets one string terminated by the parameter      */
/*      putc_low        puts one character                               */
/*      puts_low        puts one string                                  */
/*      look_low        determines destination number given station name */
int getc_low(int dest);
int gets_low(int dest,int term,char *string);
int putc_low(int dest,int c);
int puts_low(int dest,char *string);
int look_low(char *stn);

/*
 String return functions
------------------------- */
/*  In all cases the function points to string containing the answer */
/*      stnstr          returns the station string for the given ID number */
/*      stnlstr         returns the long station for the given ID number   */
/*      messtr          return the message type for the given type number  */
char *stnstr(int n,char *string);
char *stnlstr(int n,char *string);
char *messtr(int n,char *string);
```

3. COM.C

```
#define COM_VERSION     "V17Jun93.01"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <bios.h>
#include "com.h"

/*
 Miscellaneous definitions
--------------------------- */
#define DEFTIME         2               /* Default timeout is 2 s            */
#define DEFMAX          100             /* Default number of maximum errors  */
#define DEFCONSEC       10              /* Default number consecutive errors */
#define NBAUD           8               /* Number of valid baud rate keywords */
#define NPORT           MAXCOM+1        /* Number of valid port keywords     */
#define NLOW            MAXCOM          /* Number of valid low level ports   */

/*
 Serial configuration file keywords
------------------------------------ */
#define ENDFILE         -2              /* End of file in configuration file */
#define ERRLINE         -1              /* Error in configuration file       */
#define FROM            0               /* Local station name                */
#define BOARD_TYPE      1               /* Serial board type (std/digiboard) */
#define MAX_ERROR       2               /* Max number of errors before exit  */
#define TO              3               /* Introduces high-level link        */
#define LOW_LEVEL       4               /* Introduces low-level link         */
#define PORT            5               /* Port selection (COM1-COMA)        */
#define BAUD            6               /* Baud rate selection               */
#define PARITY          7               /* Parity type                       */
#define STOP            8               /* Number of stop bits               */
#define BITS            9               /* Number of bits per character      */
#define TIMEOUT         10              /* Number of bits per character      */
#define CONSECUTIVE     11              /* Number of consecutive errors      */

#define NDEF            12              /* Number of keywords                */


/*
 Serial port definitions
------------------------- */
#define NOPORT  0x00            /* No communication port           */
#define COM1    0x01                    /* COM1 to ...             */
#define COM2    0x02
#define COM3    0x03
#define COM4    0x04
#define COM5    0x05
#define COM6    0x06
#define COM7    0x07
#define COM8    0x08
#define COM9    0x09
#define COMA    0x0A                    /* ... COMA (COM10)        */
#define MAXCOM  COMA                    /* Maximum number of comm port */
#define BITS5   0x00            /* 5 bits per character            */
#define BITS6   0x01                    /* 6 bits                  */
#define BITS7   0x02                    /* 7 bits                  */
#define BITS8   0x03                    /* 8 bits                  */
#define STOP1   0x00            /* 1 stop bit                      */
#define STOP2   0x04                    /* 2 stop bits             */
#define NOPAR   0x00            /* No parity                       */
#define PARODD  0x08                    /* Odd parity              */
#define PAREVN  0x18                    /* Even parity             */
```

```
#define B110    0x00            /* Baud rate 110 bps                */
#define B150    0x20                    /* 150 bps                  */
#define B300    0x40                    /* 300 bps                  */
#define B600    0x60                    /* 600 bps                  */
#define B1200   0x80                    /* 1200 bps                 */
#define B2400   0xA0                    /* 2400 bps                 */
#define B4800   0xC0                    /* 4800 bps                 */
#define B9600   0xE0                    /* 9600 bps                 */

/*
 States for Receiver/Transmitter
-------------------------------- */
#define READY           0               /* Idle conditions          */
#define NAK_SENT        1               /* NAK sent, await retransmit  */
#define MSG_SENT        2               /* Message sent, await ACK     */
#define MSG_ACK         3               /* Message & ACK sent,await ACK */

/*
 "getline" return values
----------------------- */
#define NO_LINE         0               /* No line available        */
#define AVAIL_LINE      1               /* Line available           */

/*
 Values used for board types
--------------------------- */
#define STANDARD        0               /* Standard COM3/4 addresses & IRQs  */
#define DIGIBOARD       1               /* Digiboard COM3/4 addresses & IRQS */

/*
 States for parsing configuration file
------------------------------------- */
#define START           0               /* Start state              */
#define FROM_OK         1               /* FROM keyword valid       */
#define INTRO_OK        2               /* TO/LOW_LEVEL keyword valid  */

/*
 Communications error definitions
-------------------------------- */
#define NOERR   0               /* No error */
#define CPTACK  1               /* Ack corrupted                    */
#define CPTNAK  2               /* Nak corrupted                    */
#define CPTRXA  3               /* Receive message or ack/nak corrupted */
#define CPTTXA  4               /* Transmit message or ack corrupted    */
#define CPTTXM  5               /* Transmit message corrupted       */
#define EXTACK  6               /* Extra ack received               */
#define HEADER  7               /* Header too short                 */
#define ILCHAR  8               /* Illegal character in checksum    */
#define ILACK   9               /* Illegal ACK/NAK                  */
#define LSTACK  10              /* Ack lost, duplicate message      */
#define LSTNAK  11              /* Nak lost                         */
#define LSTRXM  12              /* Receive message lost             */
#define LSTTXM  13              /* Transmit message lost            */
#define NOCLOS  14              /* No closing bracket               */
#define NOHCHK  15              /* No trailing "h" or "H" on checksum  */
#define NOOPEN  16              /* No opening bracket               */
#define NOSEM1  17              /* No semicolon before message type */
#define NOSEM2  18              /* No semicolon after message type  */
#define NOSEPR  19              /* No from/to separator             */
#define BADCHK  20              /* Checksum failure, should be %.2X */
#define BADFRM  21              /* Bad FROM station, was "%s"       */
#define WRGFRM  22              /* Wrong FROM station, was "%s"     */
#define BADTO   23              /* Bad TO station, was "%s"         */
#define WRGTO   24              /* Wrong TO station, was "%s"       */
#define BADTYP  25              /* Bad message type, was "%s"       */
#define NERROR  26              /* Number of errors */
```

```
/* Station names used in the message headers */
char stnnam[NSTATION][LENSTN+1]={SNAMES};

/* Long station names used in configuration file */
char stntit[NSTATION][15]={LNAMES};

/* Message types used in the message headers */
char mesnam[NMESSAGE][LENMSG+1]={MNAMES};

/* Error messages */
char errtit[][50]={"No error","Ack corrupted","Nak corrupted",
                 "Transmit message or ack corrupted",
                 "Transmit message corrupted",
                 "Receive message or ack/nak corrupted","Extra ack received",
                 "Header too short","Illegal character in checksum",
                 "Illegal ACK/NAK","Ack lost, duplicate message","Nak lost",
                 "Receive message lost","Transmit message lost",
                 "No closing bracket","No trailing \"h\" or \"H\" on checksum",
                 "No opening bracket","No semicolon before message type",
                 "No semicolon after message type","No from/to separator",
                 "Checksum failure, should be ","Bad FROM station, was",
                 "Wrong FROM station, was","Bad TO station, was ",
                 "Wrong TO station, was","Bad message type, was "};


char baudtit[NBAUD][5]=                 /* Valid baud rate strings */
    {"110","150","300","600","1200","2400","4800","9600"};
int baudval[NBAUD]=                             /* Baud rate values */
    {B110,B150,B300,B600,B1200,B2400,B4800,B9600};
char deftit[NDEF][12]=  /* Valid keywords in config file */
        {"from","board_type","max_error","to","low_level","port","baud",
         "parity","stop","bits","timeout","consecutive"};
char prttit[NPORT][5]={"com1","com2",    /* Valid port names */
        "com3","com4","com5","com6","com7","com8","com9","coma","aux"};

/* Low-level link names (as given in configuration file) */
char lowtit[NLOW][50];          /* Array of names       */
int mlow;                       /* Number of names      */

/* Receive message information */
int rxtype;                     /* Receive message type         */
int rxfrom;                     /* From station of message      */
int rxto;                       /* To station of message        */
int rxcase;                     /* Case of 'H' for ack/ACK      */
char rxdata[220];               /* Message                      */

/* Queue used by get_com when 2 items are returned (ex COMM_ERR & VALID_MSG) */
int qflag;                      /* Queue flag 0=empty, 1=full   */
int qrxtyp;                     /* Message type                 */
int qrxfrm;                     /* From station of message      */
char qrxdat[220];               /* Message                      */

/* Communications error variables */
int errnum;                     /* Error number for bad msg             */
char errpar[220];               /* Error string parameter for bad msg   */
int errval;                     /* Error number parameter for bad msg   */
int errcnt;                     /* Total number of communication errors */
int maxerr;                     /* Maximum number of errors before quit */

/* Serial configuration file variables  */
FILE *sercfg;           /* File stream for configuration file           */
char cfgnam[220] = {"serial.cfg"};      /* Name of configuration file   */
int nl;                 /* Line number of line being processed          */
char lstline[220];      /* Line being processed                         */
int sfrom;              /* From station number                          */
int bd_type;            /* Board type (0=std, 1=digiboard)              */
```

```
/* Structure for serial port definitions (indexed by 0..N)      */
struct s_type {
    int to;             /* Destination of the serial link           */
    int port;           /* COM port number of the serial link       */
    int set;            /* Settings of the serial link (baud, etc)  */
} s[MAXCOM];
int nd;                 /* Number of entries in structure "s"       */


/* Structure for port definitions (indexed by COM port number)  */
struct p_type {
    int state;          /* Receiver/transmitter state               */
    int dest;           /* Destination of the serial link           */
    int error;          /* Number of consecutive errors for high-level */
    int max;            /* Maximum number of consecutive errors     */
    int time;           /* Number of ticks (16.6 ticks/s) for timeout */
    int rxack;          /* State of receiver ack (0=ack, 1=ACK)     */
    int rxpnt;          /* Pointer to receive buffer                */
    char rxbuff[220];   /* Receive buffer                           */
    char rxold[220];    /* Old received message buffer              */
    int txack;          /* State of the transmit ack (0=ack, 1=ACK) */
    char txold[220];    /* Old transmitted message buffer           */
} p[MAXCOM+1];                  /* 1 extra, COM0 is not used    */

/*
 External Assembly Language Routines
------------------------------------- */

/* Serial port routines */
int open_ser(int port,int type);    /* Initialize serial ports, set up ints */
int close_ser(int port);            /* Disable serial port interrupts       */
int stat_ser(void);                 /* Determine serial port status         */
int set_ser(int port,int parm);     /* Set baud rate, etc for serial port   */
int read_ser(int port);             /* Read a character from serial port    */
char *ver_ser(void);                /* Return version string                */
int write_ser(int port,int ich);    /* Send a character to serial port      */

/* Control-Break/Control-C ISR routines */
void open_break(void);          /* Initialize, set up trap for Ctl-C/Break   */
void close_break(void);         /* Disable trapping of Ctl-C/Break           */
int press_break(void);          /* See if break pressed (0=no, non-zero=yes) */

/* Timer tick ISR routines */
void open_time(void);               /* Initialize timers, set up ints    */
void close_time(void);              /* Disable timers                    */
int set_time(int timer,int tick);   /* Set countdown timer               */
int chk_time(int timer);            /* Check if timeout (0=timeout)      */

/* Critical error handler routines */
void open_crit(void);           /* Initialize and trap critical errors  */
void close_crit(void);          /* Disable trapping of critical errors  */


/*
 Internal routines
------------------- */
int baudmatch(char *string);    /* Determine baud rate from a string         */
void cfgerror(char *string);    /* Output error message from configuration   */
int cfgline(char *string);      /* Get a non-blank line from config file     */
int chk_error(int *dest);       /* See if any error count has exceeded max   */
char *errstr(char *string);     /* Returns the error string for an error #   */
int getline(int port);          /* Get a line from a serial port             */
int getmess(int port);          /* Get a message from a serial port          */
int lowindex(char *string);     /* Get an index value for a low-level port   */
int messtype(char *string);     /* Determine message type from a string      */
void parsemsg(int port);        /* Parse the received message                */
```

```
int prtmatch(char *string);      /* Determine port number from a string     */
int read_config(void);           /* Read configuration from config file     */
void sendack(int port);          /* Send the ack/ACK message to a port      */
void sendnak(int port);          /* Send NAK message to a port              */
void sendstr(int port,char *string);     /* Send a string to a port         */
int station(char *string);       /* Determine station number from short name */
int stnmatch(char *string);      /* Determine station number from long name  */
void strip(char *sting);         /* Remove leading and trailing blanks      */



/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - High-level Communication Routines - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - (declared in "com.h") - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

/*=========================================================================*/
/*                              open_com                                    */
/*                                                                          */
/* Description: Opens all high and low-level communications including setting */
/*              up for control-C and critical error trapping.  Reads in      */
/*              all the configuration information as well.                   */
/*                                                                          */
/* Returns:     (int)           Station number of local station.  If an error */
/*                              occurred, BAD_STATION is returned.           */
/* In:          -                                                           */
/* Out:         -                                                           */
/*-------------------------------------------------------------------------*/

int open_com(void)
{
    int i;                       /* Integer index variable */
    errcnt = 0;                  /* Initialize error reporting */
    errnum = NOERR;
    qflag = 0;                   /* Initialize queue as empty */
    mlow = 0;                    /* Initialize number of low-level ports */
    for (i=0;i<MAXCOM;i++) {     /* Initialize port structure */
        p[i].error = 0;
        p[i].rxpnt = 0;
        p[i].rxack = 0;
        p[i].txack = 0;
        p[i].rxold[0] = '\0';
        p[i].txold[0] = '\0';
    }
    if (read_config() != 0) return BAD_STATION;        /* Read config file */

    if (bd_type == DIGIBOARD) {
        printf("<< Communications hardware: Digiboard");
    } else {
        printf("<< Communications hardware: Standard");
    }
    printf("   Configuration file: %s >>\n",cfgnam);
    printf("<< Software: COM.H=%s, COM.C=%s, SERIAL.ASM=%s >>\n",
            HEAD_VERSION,COM_VERSION,ver_ser());

    open_crit();                 /* Enable trapping of critical errors */
    open_break();                /* Enable trapping of Control-C/Break */
    open_time();                 /* Enable timers */
    for (i=0;i<nd;i++) {         /* Open all serial ports from config file */
        if (open_ser(s[i].port,bd_type) != 0) {
            printf("Error in opening serial port %d\n",s[i].port);
            close_com();
            return BAD_STATION;
        }
```

```
        p[s[i].port].state = READY;
    }
    for (i=0;i<nd;i++) set_ser(s[i].port,s[i].set);     /* Setup serial ports */
    return sfrom;                   /* Return local station number */
}


/*===========================================================================*/
/*                              get_com                                      */
/*                                                                           */
/* Description: Gets a message - checks all ports for an outstanding message */
/*              also checks for if errors occurred or if control-C/break has */
/*              been pressed.  Occasionally a communications error occurs    */
/*              while a valid message is received - when this happens the    */
/*              errors is returned first and the message is queued for the   */
/*              next call.                                                   */
/*                                                                           */
/* Returns:     (int)           NO_MESSAGE - no message available            */
/*                              VALID_MSG - valid message returned           */
/*                              COMM_ERR - communications error occurred     */
/*                              QUIT - terminal condition occurred           */
/*                                                                           */
/* In:          -                                                            */
/* Out:         (int *ctype)    message type for VALID_MSG, exit type for    */
/*                              QUIT, error number for COMM_ERR              */
/*              (int *cfrom)    source of message, not used by QUIT except   */
/*                              for excessive consecutive errors             */
/*              (char *cdata)   message data for VALID_MSG or error message  */
/*                              for COMM_ERR, otherwise not used             */
/*---------------------------------------------------------------------------*/

int get_com(int *ctype, int *cfrom, char *cdata)
{
    int i,j;                        /* Integer index variables */
    int et,ef;                      /* Error type */
    if (press_break() != 0) {       /* Check if break has been pressed */
        *ctype = BREAK;
        return QUIT;
    }
    if ((et=chk_error(&ef)) != NO_ERROR) {      /* Check if max error occur */
        *ctype = et;
        *cfrom = ef;
        return QUIT;
    }
    if (qflag == 1) {               /* See if there is a message waiting */
        *ctype = qrxtyp;
        *cfrom = qrxfrm;
        strcpy(cdata,qrxdat);
        qflag = 0;
        return VALID_MSG;
    }
    for (i=0;i<nd;i++) {            /* Check all high-level for a message */
        if (s[i].to < LOW_BASE) {
            if ((j=getmess(s[i].port)) != NO_MESSAGE) {
                *ctype = rxtype;
                *cfrom = rxfrom;
                strcpy(cdata,rxdata);
                if (j == COMM_ERR) {
                    if (rxtype != BAD_MESSAGE) {        /* If error occurred, but   */
                        qrxtyp = rxtype;                /* there is a valid message */
                        qrxfrm = rxfrom;                /* put it in the queue      */
                        strcpy(qrxdat,rxdata);
                        qflag = 1;
                    }
                    *ctype = errnum;
                    *cfrom = s[i].to;
```

```
                errrstr(cdata);
                return COMM_ERR;
            } else {
                return VALID_MSG;
            }
        }
    }
  }
  return NO_MESSAGE;
}


/*===========================================================================*/
/*                              send_com                                     */
/*                                                                           */
/* Description: Sends one message to the selected destination - formats the */
/*              message, sets up the checksum and ensures reliable transfer  */
/*              through ack/nak and timeouts.                                */
/*                                                                           */
/* Returns:     (int)           0 if no error occurred                       */
/*                              1 if illegal port or if port not ready       */
/* In:          (int dest)      destination station number                   */
/*              (int mtype)     message type                                 */
/*              (char *string)  message data                                 */
/* Out:         -                                                            */
/*---------------------------------------------------------------------------*/

int send_com(int dest,int mtype,char *string)
{
    int i,n;                    /* Integer index variables */
    char str[220];              /* Used to assemble outgoing message string */
    char cc[3];                 /* Used to hold the hexadecimal checksum */
    int port;                   /* Comm port number (1=COM1, 10=COMA) */
    int chk;                    /* Checksum */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;       /* Find "to" port */
    if (n==nd) return 1;
    port = s[n].port;

    str[0] = '[';               /* Set up header */
    str[1] = '\0';              /* Message of the form: */
    strcat(str,stnnam[sfrom]);  /* [ffff>tttt;mmmmm;XXh] dddddddd..<CR><LF> */
    strcat(str,">");            /* where:      ffff is the from station      */
    strcat(str,stnnam[s[n].to]);        /*     tttt is the to station        */
    strcat(str,";");                    /*     mmmmm is the message type     */
    strcat(str,mesnam[mtype]);          /*     XX is the hex checksum        */
    if (p[port].txack == 0) {           /*     h is sent to get "ack"        */
        strcat(str,";XXh]");            /*     H is sent to get "ACK"        */
    } else {                            /*     dddddddd is the message data */
        strcat(str,";XXH]");
    }
    p[port].txack ^= 1;
    if (string[0] != '\0') {            /* Put in the space if there is data */
        strcat(str," ");
        strcat(str,string);
    }
    chk = 0;                            /* Compute the checksum */
    for (i=0;i<strlen(str);i++) if ((i<18) || (i>20)) chk += str[i];
    chk &= 0xFF;
    sprintf(cc,"%.2X",chk);
    str[18] = cc[0];
    str[19] = cc[1];
    strcat(str,"\r\n");
    if (p[port].state != READY) {       /* Ensure port is ready */
        p[port].txack ^= 1;                     /* Otherwise error & exit */
        return 1;
```

```
    } else {
        strcpy(p[port].txold,str);      /* Save the string for retransmit */
        sendstr(port,str);
        set_time(port,p[port].time);
        p[port].state = MSG_SENT;
    }
    return 0;
}


/*==========================================================================*/
/*                              look_com                                    */
/*                                                                          */
/* Description: Determine the port number given the long station name       */
/*                                                                          */
/* Returns:     (int)           port number, BAD_STATION if not valid       */
/* In:          (char *stn)     pointer to string with long station name    */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int look_com(char *stn)
{
    int i,j;            /* Integer index variables */

        /* Find the matching long station name */
    for (i=0;i<NSTATION;i++) if (strcmpi(stn,stntit[i]) == 0) break;
    if (i == NSTATION) return BAD_STATION;
        /* Find the port with that station number */
    for (j=0;j<nd;j++) if (s[j].to == i) return i;
    return BAD_STATION;
}


/*==========================================================================*/
/*                              ready_com                                   */
/*                                                                          */
/* Description: Checks to see if a link is ready for sending.               */
/*                                                                          */
/* Returns:     (int)           0 if ready                                  */
/*                              1 if still transmitting last message or bad  */
/*                                port number                               */
/* In:          (int dest)      destination station number                  */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int ready_com(int dest)
{
    int n;                      /* Integer index variables */
    int port;                   /* Comm port number (1=COM1, 10=COMA) */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find "to" port */
    if (n==nd) return 1;
    port = s[n].port;
    if (p[port].state != READY) {       /* Ensure port is ready */
        return 1;
    }
    return 0;
}


/*==========================================================================*/
/*                              config_com                                  */
/*                                                                          */
/* Description: Overrides the default (SERIAL.CFG) of the configuration file. */
/*                                                                          */
/* Returns:     -                                                           */
```

```
/* In:          (char *string)  configuration file name                  */
/* Out:         -                                                        */
/*----------------------------------------------------------------------*/

void config_com(char *string)
{
   strcpy(cfgnam,string);
   return;
}


/*======================================================================*/
/*                              flush_com                               */
/*                                                                      */
/* Description: Resets channel and associate errors.                    */
/*                                                                      */
/* Returns:     (int)           0 if ready                              */
/*                              1 if bad port number                    */
/* In:          (int dest)      destination station number              */
/* Out:         -                                                       */
/*----------------------------------------------------------------------*/

int flush_com(int dest)
{
   int n;                       /* Integer index variables */
   int port;                    /* Comm port number (1=COM1, 10=COMA) */

   for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find "to" port */
   if (n==nd) return 1;
   port = s[n].port;
   p[port].state = READY;
   p[port].error = 0;
   p[port].rxpnt = 0;
   p[port].rxack = 0;
   p[port].txack = 0;
   p[port].rxold[0] = '\0';
   p[port].txold[0] = '\0';
   errcnt = 0;
   return 0;
}


/*======================================================================*/
/*                              close_com                               */
/*                                                                      */
/* Description: Closes all high and low-level communications including the */
/*              restoration of all interrupt vectors for the serial ports, */
/*              control-C/break interrupts and critical error traps.    */
/*                                                                      */
/* Returns:     -                                                       */
/* In:          -                                                       */
/* Out:         -                                                       */
/*----------------------------------------------------------------------*/

void close_com(void)
{
   int i;               /* Integer index variable */

   for (i=0;i<nd;i++) close_ser(s[i].port);    /* Close serial ports */
   close_time();                               /* Close timers */
   close_break();                              /* Close Control-C/Break */
   close_crit();                               /* Close critical errors */
   return;
}
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - Low-level Communication Routines- - - - - - - - - - -*/
/* - - - - - - - - - - - - - - (declared in "com.h") - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/


/*==========================================================================*/
/*                              getc_low                                    */
/*                                                                          */
/* Description: Gets a character from a link using destination station number */
/*                                                                          */
/* Returns:     (int)           character if available                      */
/*                              NO_DATA is none available                   */
/*                              BAD_DEST if destination number is invalid   */
/* In:          (int dest)      destination station number (this number is */
/*                              obtained through "look_low")                */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int getc_low(int dest)
{
    int i,n;            /* Integer index variables */
    int c;              /* Character from the port */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find port number */
    if (n==nd) return BAD_DEST;
    if (p[s[n].port].rxpnt != 0) {              /* Get char from buffer */
        c=p[s[n].port].rxbuff[0];
        for (i=1;i<p[s[n].port].rxpnt;i++) p[s[n].port].rxbuff[i-1]=p[s[n].port].rxbuff[i];
        p[s[n].port].rxpnt--;
    } else {
        if ((c=read_ser(s[n].port))==-1) return NO_DATA;
    }
    return c;
}


/*==========================================================================*/
/*                              gets_low                                    */
/*                                                                          */
/* Description: Gets a terminated string from a link using the destination  */
/*              station number                                              */
/*                                                                          */
/* Returns:     (int)           ALL_OK - if string is returned              */
/*                              NO_DATA - if no data available              */
/*                              BAD_DEST - if destination number is invalid */
/* In:          (int dest)      destination station number (this number is  */
/*                              obtained through "look_low")                */
/*              (int term)      string termination character                */
/*              (char *string)  pointer to buffer to receive the string     */
/* Out:         (char *string)  pointer to string containing the received   */
/*                              string                                      */
/*--------------------------------------------------------------------------*/

int gets_low(int dest,int term,char *string)
{
    int n;              /* Integer index variable */
    int c;              /* Character from the port */
    int np;             /* Port number */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find port */
    if (n==nd) return BAD_DEST;
    np = s[n].port;
    if ((c=read_ser(np))==-1) return NO_DATA;           /* See if data avail */
```

```
    while (c!=term) {                                   /* Get data till term */
        if (p[np].rxpnt > 200) p[np].rxpnt = 200;       /* No more than 200 */
        p[np].rxbuff[p[np].rxpnt] = c;
        p[np].rxpnt++;
        if ((c=read_ser(np))==-1) return NO_DATA;       /* Data still avail? */
    }
    p[np].rxbuff[p[np].rxpnt] = '\0';           /* Terminate and save string */
    strcpy(string,p[np].rxbuff);
    p[np].rxpnt = 0;
    return ALL_OK;
}


/*========================================================================*/
/*                              putc_low                                  */
/*                                                                        */
/* Description: Send a character to a link using destination station number */
/*                                                                        */
/* Returns:     (int)           ALL_OK - if character is sent             */
/*                              BAD_DEST - if destination number is invalid */
/* In:          (int dest)      destination station number (this number is */
/*                              obtained through "look_low")              */
/*              (int c)         character to be sent                      */
/* Out:         -                                                         */
/*------------------------------------------------------------------------*/

int putc_low(int dest,int c)
{
    int n;              /* Integer index variable */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find port number */
    if (n==nd) return BAD_DEST;
    write_ser(s[n].port,c);                     /* Send character */
    return ALL_OK;
}


/*========================================================================*/
/*                              puts_low                                  */
/*                                                                        */
/* Description: Sends a string to a link using destination station number */
/*                                                                        */
/* Returns:     (int)           ALL_OK - if string is sent                */
/*                              BAD_DEST - if destination number is invalid */
/* In:          (int dest)      destination station number (this number is */
/*                              obtained through "look_low")              */
/*              (char *string)  pointer to string to be sent              */
/* Out:         -                                                         */
/*------------------------------------------------------------------------*/

int puts_low(int dest,char *string)
{
    int n;              /* Integer index variable */

    for (n=0;n<nd;n++) if (s[n].to==dest) break;        /* Find port number */
    if (n==nd) return BAD_DEST;
    sendstr(s[n].port,string);                  /* Send string */
    return ALL_OK;
}


/*========================================================================*/
/*                              look_low                                  */
/*                                                                        */
/* Description: Determines station number given the low-level station name */
/*                                                                        */
```

```
/* Returns:     (int)           destination station number                 */
/*                              BAD_STATION if invalid name                */
/* In:          (char *stn)     pointer to string containing the station name */
/* Out:         -                                                          */
/*-------------------------------------------------------------------------*/

int look_low(char *stn)
{
    int i;              /* Integer index variable */

    for (i=0;i<mlow;i++) if (strcmpi(stn,lowtit[i]) == 0) return i+LOW_BASE;
    return BAD_STATION;
}


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - -String Return Functions- - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - (declared in "com.h") - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/


/*=========================================================================*/
/*                              errstr                                     */
/*                                                                         */
/* Description: Returns a string with the error message for the last error */
/*                                                                         */
/* Returns:     (char *)        pointer to string containing error message */
/* In:          (char *string)  pointer to buffer to receive the string    */
/* Out:         (char *string)  pointer to string containing error message */
/*-------------------------------------------------------------------------*/

char *errstr(char *string)
{
    char sval[10];              /* Temporary string for formatting */

    strcpy(string,errtit[errnum]);              /* Save error text */
    switch (errnum) {
        case BADCHK:                            /* Add checksum parameter */
            sprintf(sval,"%.2Xh",errval);
            strcat(string,sval);
            break;
        case BADFRM:
        case WRGFRM:
        case BADTO:
        case WRGTO:
        case BADTYP:
            strcat(string,errpar);              /* Add string parameter */
            break;
        default:
            break;
    }
    return string;
}


/*=========================================================================*/
/*                              stnstr                                     */
/*                                                                         */
/* Description: Provide station name given the station number              */
/*                                                                         */
/* Returns:     (char *)        pointer to string with station name        */
/*                                                                         */
/* In:          (int n)         station number                             */
```

```
/*              (char *string)  pointer to buffer to receive the string       */
/* Out:         (char *string)  pointer to string with station name           */
/*--------------------------------------------------------------------------*/

char *stnstr(int n, char *string)
{
    strcpy(string,stnnam[n]);
    return string;
}


/*==========================================================================*/
/*                              stnlstr                                      */
/*                                                                           */
/* Description: Provide long station name given the station number          */
/*                                                                           */
/* Returns:     (char *)        pointer to string with long station name     */
/*                                                                           */
/* In:          (int n)         station number                               */
/*              (char *string)  pointer to buffer to receive the string      */
/* Out:         (char *string)  pointer to string with long station name     */
/*--------------------------------------------------------------------------*/

char *stnlstr(int n, char *string)
{
    strcpy(string,stntit[n]);
    return string;
}


/*==========================================================================*/
/*                              messtr                                       */
/*                                                                           */
/* Description: Provide message type string given the message type number   */
/*                                                                           */
/* Returns:     (char *)        pointer to string with message type          */
/*                                                                           */
/* In:          (int n)         message type number                          */
/*              (char *string)  pointer to buffer to receive the string      */
/* Out:         (char *string)  pointer to string with message type          */
/*--------------------------------------------------------------------------*/

char *messtr(int n, char *string)
{
    strcpy(string,mesnam[n]);
    return string;
}


/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - Internal Routines - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

/*==========================================================================*/
/*                              baudmatch                                    */
/*                                                                           */
/* Description: Determines the baud rate by matching a string with the valid */
/*              values                                                       */
/*                                                                           */
/* Returns:     (int)           Baud rate in bps (0 indicates invalid string) */
/* In:          (char *string)  Pointer to baud rate string                  */
/* Out:         -                                                            */
/*--------------------------------------------------------------------------*/
```

```
int baudmatch(char *string)
{
    int i;              /* Integer index variable */

    for (i=0;i<NBAUD;i++) if (strcmp(string,baudtit[i])==0) return baudval[i];
    return 0;
}


/*==========================================================================*/
/*                            cfgerror                                      */
/*                                                                          */
/* Description: Outputs an error message for the configuration file including */
/*              the line number and the line.  Closes configuration file    */
/*                                                                          */
/* Returns:     -                                                           */
/* In:          (char *string)  Error message string                        */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

void cfgerror(char *string)
{
    int i;
    printf("%s in line %d of %s\n",string,nl,cfgnam);   /* Error in line # */
    lstline[strlen(lstline)-1] = '\0';
    printf("%s\n",lstline);                              /* Output line */
    printf("Debug: ");
    for (i=0;i<20;i++) printf("%02X ",lstline[i]);
    printf("\nDebug: ");
    for (i=20;i<40;i++) printf("%02X ",lstline[i]);
    printf("\n");
    fclose(sercfg);                                      /* Close config file */
    return;
}


/*==========================================================================*/
/*                            cfgline                                       */
/*                                                                          */
/* Description: Gets a non-blank line from configuration file               */
/*              Line must be of the form    <keyword> = <value_string>      */
/*                                                                          */
/* Returns:     (int)           Keyword number (see defines)                */
/*                              ENDFILE for end of file                     */
/*                              ERRLINE for unrecognized line               */
/* In:          -                                                           */
/* Out:         (char *string)  The value_string                            */
/*--------------------------------------------------------------------------*/

int cfgline(char *string)
{
    int i;                      /* Integer index variable */
    char line[220];             /* String holding line read from config file */
    char c[220];                /* String to hold the keyword */

    do {
        if (fgets(line,220,sercfg)==NULL) return ENDFILE;     /* End of file */
        strcpy(lstline,line);           /* Save line for error message */
        nl++;
        line[strlen(line)-1] = '\0';    /* Remove '\n' and terminate line */
        strip(line);                    /* Romove leading and trailing blanks */
        if (line[0]==';') line[0]='\0';         /* Ignore comment lines */
    } while (line[0] == '\0');
    for (i=0;i<strlen(line);i++) line[i] = tolower(line[i]);    /* Lower case */
    for (i=0;i<strlen(line);i++) if (line[i]=='=') break;       /* Find '=' */
    if (i==strlen(line)) return ERRLINE;
```

```
    strncpy(c,line,i);              /* Extract, terminate and strip keyword */
    c[i] = '\0';
    strip(c);
    strcpy(string,&line[i+1]);      /* Extract and strip value string */
    strip(string);
    for (i=0;i<NDEF;i++) if (strcmp(c,deftit[i])==0) break;   /* Find keyword */
    if (i == NDEF) return ERRLINE;
    return i;
}


/*==========================================================================*/
/*                              chk_error                                   */
/*                                                                          */
/* Description: Checks if any error count (total or consecutive on any port */
/*              has exceeded the maximums                                   */
/*                                                                          */
/* Returns:     (int)           TOTAL - total number of errors exceeded     */
/*                              CONSEC - max consecutive errors on 1 port   */
/*                              NO_ERROR - no errors                        */
/* In:          (*int dest)     Station causing error (when valid)          */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int chk_error(int *dest)
{
    int i;              /* Integer index variable */

    if (errcnt > maxerr) {      /* Check for total errors */
        *dest = UNKNOWN_ID;
        return TOTAL;
    }
    for (i=0;i<nd;i++) {        /* Check for consecutive errors on any link */
        if (p[s[i].port].error >= p[s[i].port].max) {
            *dest = s[i].to;
            return CONSEC;
        }
    }
    *dest = UNKNOWN_ID;
    return NO_ERROR;
}


/*==========================================================================*/
/*                              getline                                     */
/*                                                                          */
/* Description: Gets a line terminated by CR from a serial port.  Control   */
/*              characters are discarded.  Line available in "p[port].rxbuff" */
/*                                                                          */
/* Returns:     (int)           AVAIL_LINE - "p[port].rxbuff" has the line  */
/*                              NO_LINE - no line available                 */
/* In:          (int port)      Port number (1=COM1 to 10=COMA)             */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int getline(int port)
{
    int c;              /* Character read from port */

    if ((c=read_ser(port))==-1) return NO_LINE; /* See if char avail */
    while (c!=0x0D) {                           /* Until <CR> */
        if (c >= 0x20) {                        /* Ignore cntl chars */
            if (p[port].rxpnt > 200) p[port].rxpnt = 200;       /* Max 200 */
            p[port].rxbuff[p[port].rxpnt] = c;
            p[port].rxpnt++;
        }
```

```
        if ((c=read_ser(port))==-1) return NO_LINE;     /* Any avail still ? */
    }
    p[port].rxbuff[p[port].rxpnt] = '\0';
    return AVAIL_LINE;
}


/*==========================================================================*/
/*                              getmess                                     */
/*                                                                          */
/* Description: Gets a message from a serial port.  Controls the ACK/NAK    */
/*              handshaking and error detection.  Message details are as    */
/*              described for "parsemsg"                                    */
/*                                                                          */
/* Returns:     (int)           VALID_MSG - a valid message is available    */
/*                              NO_MESSAGE - no message available           */
/*                              COMM_ERROR - communication error occured    */
/* In:          (int port)      Port number (1=COM1, 10=COMA)               */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int getmess(int port)
{
    struct p_type *pp;          /* Pointer to port structure */

    pp = &p[port];              /* Get pointer to port structure */
    switch (pp->state) {
    /* Ready state - no outstanding messages, acks or timeouts */
        case READY:
            if (getline(port) == NO_LINE) return NO_MESSAGE;
            parsemsg(port);
            if (rxtype == BAD_MESSAGE) {        /* Bad message => nak */
                sendnak(port);
                set_time(port,p[port].time);
                pp->state = NAK_SENT;
                errnum = CPTRXA;
                errcnt++;
                pp->error++;
                return COMM_ERR;
            } else if (rxtype == NAK) {         /* Nak is extra */
                pp->rxack ^= 1;
                sendack(port);
                errnum = CPTACK;
                errcnt++;
                pp->error++;
                rxtype = BAD_MESSAGE;
                return COMM_ERR;
            } else if (rxtype == ACK) {         /* Ack is extra */
                errnum = EXTACK;
                errcnt++;
                pp->error++;
                rxtype = BAD_MESSAGE;
                return COMM_ERR;
            } else if (rxcase != pp->rxack) {           /* Out of msg sync */
                if (strcmp(pp->rxbuff,pp->rxold) != 0) {        /* New msg */
                    pp->rxack ^= 1;
                    sendack(port);
                    strcpy(pp->rxold,pp->rxbuff);
                    errnum = LSTRXM;
                    errcnt++;
                    pp->error++;
                    return COMM_ERR;
                } else {
                    pp->rxack ^= 1;                             /* Old msg */
                    sendack(port);
                    errnum = LSTACK;
```

```
            rxtype = BAD_MESSAGE;
            errcnt++;
            pp->error++;
            return COMM_ERR;
        }
    } else {
        strcpy(pp->rxold,pp->rxbuff);          /* Valid message */
        sendack(port);
    }
    break;
/* Nak sent state - awaiting retranmission of message or ack/nak */
  case NAK_SENT:
    if (chk_time(port) == 0) {          /* Timeout => retransmit nak */
        sendnak(port);
        set_time(port,p[port].time);
        errnum = LSTNAK;
        rxtype = BAD_MESSAGE;
        errcnt++;
        pp->error++;
        return COMM_ERR;
    }
    if (getline(port) == NO_LINE) return NO_MESSAGE;
    parsemsg(port);
    if (rxtype == BAD_MESSAGE) {        /* Bad message => nak */
        sendnak(port);
        set_time(port,p[port].time);
        errnum = CPTRXA;
        errcnt++;
        pp->error++;
        return COMM_ERR;
    } else if (rxtype == NAK) {         /* Nak => retransmit nak */
        sendnak(port);
        set_time(port,p[port].time);
        errnum = CPTNAK;
        errcnt++;
        pp->error++;
        rxtype = BAD_MESSAGE;
        return COMM_ERR;
    } else if (rxtype == ACK) {         /* Extra ack */
        pp->state = READY;
        errnum = EXTACK;
        errcnt++;
        pp->error++;
        rxtype = BAD_MESSAGE;
        return COMM_ERR;
    } else if (rxcase != pp->rxack) {           /* Out of msg sync */
        if (strcmp(pp->rxbuff,pp->rxold) != 0) {        /* New msg */
            pp->state = READY;
            pp->rxack ^= 1;
            sendack(port);
            strcpy(pp->rxold,pp->rxbuff);
            errnum = LSTRXM;
            errcnt++;
            pp->error++;
            return COMM_ERR;
        } else {
            pp->state = READY;                          /* Old msg */
            pp->rxack ^= 1;
            sendack(port);
            errnum = LSTACK;
            rxtype = BAD_MESSAGE;
            errcnt++;
            pp->error++;
            return COMM_ERR;
        }
    } else {
```

```
            pp->state = READY;                /* Valid message */
            strcpy(pp->rxold,pp->rxbuff);
            sendack(port);
         }
         brek;
/* Message sent state - awaiting ack */
      case MS_SENT:
         if chk_time(port) == 0) {            /* Timeout => retransmit */
            sendstr(port,pp->txold);
            set_time(port,p[port].time);
            errnum = LSTTXM;
            rxtype = BAD_MESSAGE;
            errcnt++;
            pp->error++;
            return COMM_ERR;
         }
         if getline(port) == NO_LINE) return NO_MESSAGE;
         paremsg(port);
         if rxtype == BAD_MESSAGE) {          /* Bad message => nak */
            sendnak(port);
            set_time(port,p[port].time);
            errnum = CPTRXA;
            errcnt++;
            pp->error++;
            return COMM_ERR;
         } ese if (rxtype == NAK) {           /* Nak => retransmit */
            sendstr(port,pp->txold);
            set_time(port,p[port].time);
            errnum = CPTTXM;
            errcnt++;
            pp->error++;
            rxtype = BAD_MESSAGE;
            return COMM_ERR;
         } ele if (rxtype == ACK) {           /* Ack received */
             f (pp->txack == rxcase) {                 /* Out of msg sync */
               sendstr(port,pp->txold);
               set_time(port,p[port].time);
               errnum = LSTTXM;
               errcnt++;
               pp->error++;
               rxtype = BAD_MESSAGE;
               return COMM_ERR;
             else {
               pp->state = READY;                      /* Ack OK */
               pp->error = 0;
               return NO_MESSAGE;

         } ele if (rxcase != pp->rxack) {              /* Out of msg sync */
             f (strcmp(pp->rxbuff,pp->rxold) != 0) {          /* New msg */
               pp->state = MSG_ACK;
               pp->rxack ^= 1;
               sendack(port);
               set_time(port,p[port].time);
               strcpy(pp->rxold,pp->rxbuff);
               errnum = LSTRXM;
               errcnt++;
               pp->error++;
               return COMM_ERR;
            } else {
               pp->state = MSG_ACK;                           /* Old msg */
               pp->rxack ^= 1;
               sendack(port);
               set_time(port,p[port].time);
               errnum = LSTACK;
               rxtype = BAD_MESSAGE;
               errcnt++;
```

```
                pp->error++;
                return COMM_ERR;
            }
        } else {
            pp->state = MSG_ACK;            /* Valid message received */
            strcpy(pp->rxold,pp->rxbuff);
            sendack(port);
            set_time(port,p[port].time);
        }
        break;
/* Message and Ack transmitted, awaiting ack */
    case MSG_ACK:
        if (chk_time(port) == 0) {          /* Timeout => retransmit */
            sendstr(port,pp->txold);        /* message               */
            set_time(port,p[port].time);
            errnum = LSTTXM;
            rxtype = BAD_MESSAGE;
            errcnt++;
            pp->error++;
            return COMM_ERR;
        }
        if (getline(port) == NO_LINE) return NO_MESSAGE;
        parsemsg(port);
        if (rxtype == BAD_MESSAGE) {        /* Bad message => retransmit */
            pp->rxack ^= 1;                 /* both ack and message      */
            sendack(port);
            sendstr(port,pp->txold);
            set_time(port,p[port].time);
            errnum = CPTRXA;
            errcnt++;
            pp->error++;
            return COMM_ERR;
        } else if (rxtype == NAK) {         /* Nak => retransmit both */
            pp->rxack ^= 1;
            sendack(port);
            sendstr(port,pp->txold);
            set_time(port,p[port].time);
            errnum = CPTTXA;
            errcnt++;
            pp->error++;
            rxtype = BAD_MESSAGE;
            return COMM_ERR;
        } else if (rxtype == ACK) {         /* Ack received */
            if (pp->txack == rxcase) {              /* Out of msg sync */
                sendstr(port,pp->txold);
                set_time(port,p[port].time);
                errnum = LSTTXM;
                errcnt++;
                pp->error++;
                rxtype = BAD_MESSAGE;
                return COMM_ERR;
            } else {
                pp->state = READY;                  /* Valid ack */
                pp->error = 0;
                return NO_MESSAGE;
            }
        } else if (rxcase != pp->rxack) {           /* Out of msg sync */
            if (strcmp(pp->rxbuff,pp->rxold) != 0) {        /* New msg */
                pp->rxack ^= 1;
                sendack(port);
                set_time(port,p[port].time);
                strcpy(pp->rxold,pp->rxbuff);
                errnum = LSTRXM;
                errcnt++;
                pp->error++;
                return COMM_ERR;
```

```
                } else {
                    pp->rxack ^= 1;                           /* Old msg */
                    sendack(port);
                    set_time(port,p[port].time);
                    errnum = LSTACK;
                    rxtype = BAD_MESSAGE;
                    errcnt++;
                    pp->error++;
                    return COMM_ERR;
                }
            } else {
                strcpy(pp->rxold,pp->rxbuff);          /* Valid message */
                sendack(port);
                set_time(port,p[port].time);
            }
            break;
    }
    pp->error = 0;
    return VALID_MSG;
}


/*===========================================================================*/
/*                                lowindex                                   */
/*                                                                           */
/* Description: Determines index number for low-level port names.  All index */
/*              numbers are based on LOW_BASE and do not conflict with high- */
/*              level port numbers.                                          */
/*                                                                           */
/* Returns:     (int)           Index number for the low-level port, to be   */
/*                              used as station number in other calls.  If   */
/*                              that name has already been used, then it     */
/*                              returns BAD_STATION                          */
/* In:          (char *string)  Pointer to string with low-level port name   */
/* Out:         -                                                            */
/*---------------------------------------------------------------------------*/

int lowindex(char *string)
{
    int i;              /* Integer index variable */

        /* Check to see if name is already used */
    for (i=0;i<mlow;i++) if (strcmp(string,lowtit[i])==0) return BAD_STATION;
    strcpy(lowtit[mlow],string);
    mlow++;
    return mlow - 1 + LOW_BASE;        /* Return numbers starting at LOW_BASE */
}


/*===========================================================================*/
/*                                messtype                                   */
/*                                                                           */
/* Description: Determines the index number for the message type string      */
/*              Only the number of characters in the message type field      */
/*              are checked (LENMSG).                                        */
/*                                                                           */
/* Returns:     (int)           Index number for the message type.  If the   */
/*                              message string is not recognized, it returns */
/*                              BAD_MESSAGE                                  */
/* In:          (char *string)  Pointer to string with message type          */
/* Out:         -                                                            */
/*---------------------------------------------------------------------------*/

int messtype(char *string)
{
    int i,j;            /* Integer index variables */
```

```
    for(i=0;i<NMESSAGE;i++) {           /* Check for message type match */
        for(j=0;j<LENMSG;j++) if (string[j] != mesnam[i][j]) break;
        if (j == LENMSG) return i;
    }
    return BAD_MESSAGE;
}


/*==========================================================================*/
/*                              parsemsg                                    */
/*                                                                          */
/* Description:   Parses message stored in "p[port].rxbuff" including error */
/*              and format checking.  Source and destination stations are  */
/*              stored in "rxfrom" and "rxto" respectively.  The message type */
/*              is in "rxtype".  The string "rxdata" contains the data part */
/*              of the message.  "rxcase" contains the case of the 'H' which */
/*              indicates the case needed for the ack/ACK.                 */
/*                In the event of an error, "rxtype" is BAD_MESSAGE.  The error */
/*              number is stored in "errnum", string parameter (if required) */
/*              is stored in "errpar" and if necessary the integer parameter */
/*              is stored in "errval".                                     */
/*                Once the parsing is complete, the buffer pointer is reset */
/*                                                                          */
/* Returns:     -                                                           */
/* In:          (int port)      Port number (1=COM1, 10=COMA)               */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

void parsemsg(int port)
{
    int i;              /* Integer index variable */
    int sndchk;         /* Checksum sent with message */
    int chk;            /* Computed checksum on receive message */

    rxfrom = 0;
    rxto = 0;
    if (p[port].rxpnt == 3) {           /* 3 chars => ack, ACK or NAK */
        if (strcmp(p[port].rxbuff,"ack")==0) {
            rxtype = ACK;                       /* ack, no data */
            rxcase = 0;
            rxdata[0] = '\0';
        } else if (strcmp(p[port].rxbuff,"ACK")==0) {
            rxtype = ACK;                       /* ACK, no data */
            rxcase = 1;
            rxdata[0] = '\0';
        } else if (strcmpi(p[port].rxbuff,"nak")==0) {
            rxtype = NAK;                       /* NAK, no data */
            rxdata[0] = '\0';
        } else {
            rxtype = BAD_MESSAGE;               /* otherwise, bad message */
            errnum = ILACK;
            p[port].rxpnt = 0;
            return;
        }
    } else if (p[port].rxpnt < 22) {    /* <22 chars => header too short */
        rxtype = BAD_MESSAGE;
        errnum = HEADER;
        p[port].rxpnt = 0;
        return;
    } else {                            /* Check case of 'h' for ack/ACK */
        if (p[port].rxbuff[20] == 'h') {
            rxcase = 0;
        } else if (p[port].rxbuff[20] == 'H') {
            rxcase = 1;
        } else {
            rxtype = BAD_MESSAGE;               /* Not 'h' or 'H' => bad head */
```

```
    errnum = NOHCHK;
    p[port].rxpnt = 0;
    return;
}
    /* Make all characters lower case */
for (i=0;i<22;i++) p[port].rxbuff[i] = tolower(p[port].rxbuff[i]);
if (p[port].rxbuff[0] != '[') {
    rxtype = BAD_MESSAGE;              /* No opening bracket */
    errnum = NOOPEN;
    p[port].rxpnt = 0;
    return;
}
if (p[port].rxbuff[5] != '>') {
    rxtype = BAD_MESSAGE;              /* No separator */
    errnum = NOSEPR;
    p[port].rxpnt = 0;
    return;
}
if (p[port].rxbuff[10] != ';') {
    rxtype = BAD_MESSAGE;              /* 1st ';' separator missing */
    errnum = NOSEM1;
    p[port].rxpnt = 0;
    return;
}
if (p[port].rxbuff[17] != ';') {
    rxtype = BAD_MESSAGE;              /* 2nd ';' separator missing */
    errnum = NOSEM2;
    p[port].rxpnt = 0;
    return;
}
if (p[port].rxbuff[21] != ']') {
    rxtype = BAD_MESSAGE;              /* No closing bracket */
    errnum = NOCLOS;
    p[port].rxpnt = 0;
    return;
}
rxfrom = station(&p[port].rxbuff[1]);
if (rxfrom == BAD_STATION) {
    rxtype = BAD_MESSAGE;              /* Unrecognized from station */
    errnum = BADFRM;
    strncpy(errpar,&p[port].rxbuff[1],LENSTN);
    errpar[LENSTN] = '\0';
    p[port].rxpnt = 0;
    return;
} else if (rxfrom != p[port].dest) {
    rxtype = BAD_MESSAGE;              /* 'from' station does not  */
    errnum = WRGFRM;                   /*  match link destination  */
    strncpy(errpar,&p[port].rxbuff[1],LENSTN);
    errpar[LENSTN] = '\0';
    p[port].rxpnt = 0;
    return;
}
rxto = station(&p[port].rxbuff[6]);
if (rxto == BAD_STATION) {
    rxtype = BAD_MESSAGE;              /* Unrecognized 'to' station */
    errnum = BADTO;
    strncpy(errpar,&p[port].rxbuff[6],LENSTN);
    errpar[LENSTN] = '\0';
    p[port].rxpnt = 0;
    return;
} else if (rxto != sfrom) {
    rxtype = BAD_MESSAGE;              /* 'to' station does not */
    errnum = WRGTO;                    /*  match local station  */
    strncpy(errpar,&p[port].rxbuff[6],LENSTN);
    errpar[LENSTN] = '\0';
    p[port].rxpnt = 0;
```

```
            return;
        }
        rxtype = messtype(&p[port].rxbuff[11]);
        if (rxtype == BAD_MESSAGE) {
            rxtype = BAD_MESSAGE;               /* Unrecognized message type */
            errnum = BADTYP;
            strncpy(errpar,&p[port].rxbuff[11],LENMSG);
            errpar[LENMSG] = '\0';
            p[port].rxpnt = 0;
            return;
        }
        if ((tolower(p[port].rxbuff[18])!='x') ||
                (tolower(p[port].rxbuff[19])!='x')) {
            /* Only check checksum if field is not 'xx' or 'XX' */
            if ((!isxdigit(p[port].rxbuff[18])) ||
                (!isxdigit(p[port].rxbuff[19]))) {
                rxtype = BAD_MESSAGE;           /* Checksum isn't hexadecimal */
                errnum = ILCHAR;
                p[port].rxpnt = 0;
                return;
            }
            sscanf(&p[port].rxbuff[18],"%2x",&sndchk);  /* Get tx checksum */
            chk = 0;                            /* Compute receive checksum */
            for (i=0;i<p[port].rxpnt;i++) if ((i<18) || (i>20))
                chk += p[port].rxbuff[i];
            chk &= 0xFF;
            if (chk != sndchk) {
                rxtype = BAD_MESSAGE;           /* Checksum doesn't match */
                errnum = BADCHK;
                errval = chk;
                p[port].rxpnt = 0;
                return;
            }
        }
        if (p[port].rxpnt < 24) {
            rxdata[0] = '\0';                   /* <24 chars => no data field */
        } else {
            strcpy(rxdata,&p[port].rxbuff[23]); /* Get data field, skip blank */
        }
    }
    p[port].rxpnt = 0;                  /* Reset receive buffer pointer */
    return;
}


/*==========================================================================*/
/*                              prtmatch                                    */
/*                                                                          */
/* Description: Determines the port by matching a string with valid values  */
/*              COM1-9 are normal.  COMA is used instead of "COM10" to ensure */
/*              a constant length field.  AUX is a synonym for COM1.        */
/*                                                                          */
/* Returns:     (int)           Port number (1=COM1, 10=COMA)  If no match  */
/*                              is found, 0 is returned                     */
/* In:          (char *string)  Pointer to port string                      */
/* Out:         -                                                           */
/*--------------------------------------------------------------------------*/

int prtmatch(char *string)
{
    int i;              /* Integer index variable */
    char t[220];        /* Temporary string variable */

    strcpy(t,string);           /* Copy string to temp, remove ':' if there */
    i = strlen(t);
    if (t[i-1]==':') t[i-1] = '\0';
```

```
    for (i=0;i<NPORT;i++) if (strcmp(t,prttit[i])==0) break;
    if (i == NPORT) return 0;           /* No valid match was found */
    if (i == NPORT-1) return 1;         /* 'AUX' is changed to 'COM1' */
    return i+1;                 /* Return port number */
}


/*=========================================================================*/
/*                              read_config                                */
/*                                                                         */
/* Description: Read the configuration file to set up the port            */
/*              usage and stations names.  Sets up the serial structure "s"  */
/*              for any given link "i" with "s[i].to" as the destination   */
/*              station, "s[i].port" as the port number and "s[i].set" as the */
/*              serial port settings (baud rate etc).  "nd" contains the   */
/*              of links.  Also sets up the port structure "p" for any given */
/*              port "port" with "p[port].dest" as the destination station. */
/*                                                                         */
/* Returns:     (int)          0 if config file is ok,1 if an error occurred */
/* In:          -                                                          */
/* Out:         -                                                          */
/*-------------------------------------------------------------------------*/

int read_config(void)
{
    int i;              /* Integer index variable */
    int baud;           /* Baud rate */
    int parity;         /* Parity */
    int stop;           /* Number of stop bits */
    int bits;           /* Number of bits per character */
    int dtype;          /* Keyword type number */
    int state;          /* State of the configuration file processor */
    int itimeout;       /* Number of seconds before timeout */
    int consecut;       /* Number of consecutive errors allowed before exit */
    char parm[220];     /* String parameter for the keyword */

    nl = 0;
    lstline[0] = '\0';
    nd = 0;
    bd_type = 1;
    if ((sercfg=fopen(cfgnam,"r")) == NULL) {   /* Open configuration file */
        printf("Cannot open %s\n",cfgnam);
        return 1;
    }
    state = START;
    while ((dtype=cfgline(parm)) != ENDFILE) {
        switch (state) {
            /* Start state - waiting for FROM to specify local station */
            case START:
                if (dtype == FROM) {            /* FROM keyword */
                    if ((sfrom=stnmatch(parm)) == BAD_STATION) {
                        cfgerror("Unrecognized FROM station");
                        return 1;
                    }
                    state = FROM_OK;
                } else if (dtype == ERRLINE) {  /* Unrecognized line */
                    cfgerror("Unrecognized definition");
                    return 1;
                } else {                        /* Keyword other than FROM */
                    cfgerror("Found a definition not preceeded by FROM");
                    return 1;
                }
                break;
            /* From OK state - waiting for TO/LOW_LEVEL to intro link */
            case FROM_OK:
                maxerr = DEFMAX;
```

```
        s[nd].port = NOPORT;            /* Set default link values */
        baud = B9600;
        parity = NOPAR;
        stop = STOP1;
        bits = BITS8;
        itimeout = DEFTIME;
        consecut = DEFCONSEC;
        if (dtype == TO) {                      /* High-level link */
            if ((s[nd].to=stnmatch(parm)) == BAD_STATION) {
                cfgerror("Unrecognized TO station");
                return 1;
            }
            state = INTRO_OK;
        } else if (dtype == LOW_LEVEL) {        /* Low level link */
            if ((s[nd].to=lowindex(parm)) == BAD_STATION) {
                cfgerror("Low level port name not unique");
                return 1;
            }
            state = INTRO_OK;
        } else if (dtype == BOARD_TYPE) {       /* Specify board type */
            if (strcmp(parm,"digiboard")==0) {
                bd_type = DIGIBOARD;
            } else if (strcmp(parm,"standard")==0) {
                bd_type = STANDARD;
            } else {
                cfgerror("Unrecognized board type");
                return 1;
            }
        } else if (dtype == MAX_ERROR) {        /* Maximum errors */
            sscanf(parm,"%d",&maxerr);
            if ((maxerr < 1) || (maxerr > 30000)) {
                cfgerror("Maximum errors must be in range 1-30000");
                return 1;
            }
            break;
        } else if (dtype == ERRLINE) {          /* Unrecognized line */
            cfgerror("Unrecognized definition");
            return 1;
        } else {                                /* Other keywords */
            cfgerror("Comm parameters without TO or LOW_LEVEL");
            return 1;
        }
        break;
    /* Intro OK - waiting for comm parameters or another intro */
    case INTRO_OK:
        switch (dtype) {
            case ERRLINE:                       /* Unrecognized line */
                cfgerror("Unrecognized definition");
                return 1;
            case FROM:                          /* Extra From */
                cfgerror("Multiple FROM definition");
                return 1;
            case BOARD_TYPE:                    /* Bd type misplaced */
                cfgerror("Board type definition must follow FROM");
                return 1;
            case MAX_ERROR:                     /* Max err misplaced */
                cfgerror("Maximum error must follow FROM");
                return 1;
            case TO:
            case LOW_LEVEL:                     /* Another link intro */
                if (s[nd].port == NOPORT) {     /* PORT= is missing */
                    cfgerror("No PORT definition found");
                    return 1;
                }
                s[nd].set = baud + parity + stop + bits;
                p[s[nd].port].dest = s[nd].to;
```

```
        p[s[nd].port].time = (int)(itimeout * 16.66);
        p[s[nd].port].max = consecut;
        nd++;
        if (nd >= MAXCOM) {
            cfgerror("Maximum number of ports exceeded");
            return 1;
        }
        s[nd].port = NOPORT;    /* Set default parameters */
        baud = B9600;
        parity = NOPAR;
        stop = STOP1;
        bits = BITS8;
        itimeout = DEFTIME;
        consecut = DEFCONSEC;
        if (dtype == TO) {      /* High-level link */
            if ((s[nd].to=stnmatch(parm)) == BAD_STATION) {
                cfgerror("Unrecognized TO station");
                return 1;
            }
        } else {                /* Low-level link */
            if ((s[nd].to=lowindex(parm)) == BAD_STATION) {
                cfgerror("Low level port name not unique");
                return 1;
            }
        }
        state = INTRO_OK;
        break;
    case PORT:                  /* Define COM port to be used */
        if ((s[nd].port=prtmatch(parm)) == 0) {
            cfgerror("Unrecognized port type");
            return 1;
        }
        if (nd > 0) {           /* Check port not already use */
            for (i=0;i<nd;i++) {
                if (s[nd].port == s[i].port) {
                    cfgerror("Redefinition of serial port");
                    return 1;
                }
            }
        }
        break;
    case BAUD:                  /* Define baud rate */
        if ((baud=baudmatch(parm)) == 0) {
            cfgerror("Unrecognized baud rate");
            return 1;
        }
        break;
    case PARITY:                /* Define parity */
        if (strcmp(parm,"none")==0) {
            parity = NOPAR;
        } else if (strcmp(parm,"even")==0) {
            parity = PAREVN;
        } else if (strcmp(parm,"odd")==0) {
            parity = PARODD;
        } else {
            cfgerror("Unrecognized parity");
            return 1;
        }
        break;
    case STOP:                  /* Define number of stop bits */
        if (strcmp(parm,"1")==0) {
            stop = STOP1;
        } else if (strcmp(parm,"1.5")==0) {
            stop = STOP1;
        } else if (strcmp(parm,"2")==0) {
            stop = STOP2;
```

```
                    } else {
                        cfgerror("Unrecognized stop bits");
                        return 1;
                    }
                    break;
                case BITS:                  /* Define bits per character */
                    if (strcmp(parm,"5")==0) {
                        bits = BITS5;
                    } else if (strcmp(parm,"6")==0) {
                        bits = BITS6;
                    } else if (strcmp(parm,"7")==0) {
                        bits = BITS7;
                    } else if (strcmp(parm,"8")==0) {
                        bits = BITS8;
                    } else {
                        cfgerror("Unrecognized bits/character");
                        return 1;
                    }
                    break;
                case TIMEOUT:               /* Set timeout */
                    sscanf(parm,"%d",&itimeout);
                    if ((itimeout < 1) || (itimeout > 100)) {
                        cfgerror("Timeout must be in range 1-100");
                        return 1;
                    }
                    break;
                case CONSECUTIVE:           /* Maximum errors */
                    sscanf(parm,"%d",&consecut);
                    if ((consecut < 1) || (consecut > 10000)) {
                        cfgerror("Consecutive errors must be in range 1-10000");
                        return 1;
                    }
                    break;
            }
            break;
        }
    }
    switch (state) {
        case START:
            cfgerror("No FROM definition found");
            return 1;
            break;
        case FROM_OK:
            break;
        case INTRO_OK:
            if (s[nd].port == NOPORT) {             /* PORT= missing */
                cfgerror("No PORT definition found for last TO");
                return 1;
            }
            s[nd].set = baud + parity + stop + bits;
            p[s[nd].port].dest = s[nd].to;
            p[s[nd].port].time = (int)(itimeout * 16.66);
            p[s[nd].port].max = consecut;
            nd++;
            break;
    }
    fclose(sercfg);             /* Close file */
    return 0;
}

/*==========================================================================*/
/*                              sendack                                     */
/*                                                                          */
/* Description: Sends an ack/ACK of the appropriate case to the port        */
/*                                                                          */
/*
```

```
/* Returns:     -                                                      */
/* In:          (int)           Port number (1=COM1, 10=COMA)          */
/* Out:         -                                                      */
/*---------------------------------------------------------------------*/

void sendack(int port)
{
    if (p[port].rxack == 0) {
        sendstr(port,"ack\r\n");
    } else {
        sendstr(port,"ACK\r\n");
    }
    p[port].rxack ^= 1;              /* Toggle case for next ack/ACK */
    return;
}


/*=====================================================================*/
/*                              sendnak                                */
/*                                                                     */
/* Description: Sends a nak to the port                                */
/*                                                                     */
/* Returns:     -                                                      */
/* In:          (int)           Port number (1=COM1, 10=COMA)          */
/* Out:         -                                                      */
/*---------------------------------------------------------------------*/

void sendnak(int port)
{
    sendstr(port,"nak\r\n");
    return;
}


/*=====================================================================*/
/*                              sendstr                                */
/*                                                                     */
/* Description: Sends a string to the port                             */
/* Returns:     -                                                      */
/* In:          (int)           Port number (1=COM1, 10=COMA)          */
/*              (char *string)  Pointer to string to be sent           */
/* Out:         -                                                      */
/*---------------------------------------------------------------------*/

void sendstr(int port,char *string)
{
    int i;              /* Integer index variable */

    for(i=0;i<strlen(string);i++) write_ser(port,string[i]);
    return;
}


/*=====================================================================*/
/*                              station                                */
/*                                                                     */
/* Description: Determines the station number by matching a string with valid */
/*              values.  Only the number of characters in the from/to field   */
/*              are checked (LENSTN).                                  */
/*                                                                     */
/* Returns:     (int)           Station number.  If the string is not  */
/*                              recognized, it returns BAD_STATION     */
/* In:          (char *string)  Pointer to string with station name    */
/* Out:         -                                                      */
/*---------------------------------------------------------------------*/
```

```
int station(char *string)
{
    int i,j;            /* Integer index variables */

    for(i=0;i<NSTATION;i++) {           /* Match only LENSTR characters */
        for(j=0;j<LENSTN;j++) if (string[j] != stnnam[i][j]) break;
        if (j == LENSTN) return i;      /* Match found */
    }
    return BAD_STATION;
}


/*===========================================================================*/
/*                              stnmatch                                     */
/*                                                                           */
/* Description: Determines the station number by matching a string with the  */
/*              valid long station names (used in configuration file).       */
/*                                                                           */
/* Returns:     (int)           Station number.  If the string is not        */
/*                              recognized, it returns BAD_STATION           */
/* In:          (char *string)  Pointer to string with long station name     */
/* Out:         -                                                            */
/*---------------------------------------------------------------------------*/

int stnmatch(char *string)
{
    int i;              /* Integer index variable */

    for (i=0;i<NSTATION;i++) if (strcmp(string,stntit[i])==0) break;
    if (i == NSTATION) return BAD_STATION;      /* No match found */
    return i;
}


/*===========================================================================*/
/*                              strip                                        */
/*                                                                           */
/* Description: Removes trailing and leading blanks from a string            */
/*                                                                           */
/* Returns:     -                                                            */
/* In:          (char *string)  Pointer to string to be stripped of blanks   */
/* Out:         (char *string)  Pointer to string that has been stripped     */
/*---------------------------------------------------------------------------*/

void strip(char *string)
{
    int i;              /* Integer index variable */
    char t[220];        /* Temporary working string */

        /* Check (from beginning) for non-blank character */
    for (i=0;i<strlen(string);i++) if (!isspace(string[i])) break;
    if (i == strlen(string)) {                  /* Blank string */
        string[0] = '\0';
        return;
    }
    strcpy(t,&string[i]);                       /* Remove leading spaces */
        /* Check (from end) for non-blank character */
    for (i=strlen(t)-1;i>0;i--) if (!isspace(string[i])) break;
    strncpy(string,t,i+1);                      /* Remove trailing spaces */
    string[i+1] = '\0';
    return;
}
```

Appendix E

Real-time Software Listing

1. Introduction

In this appendix, the assembly language file SERIAL.ASM is listed. This file includes all of the real-time software used by the communications software. This appendix does not cover the C-language portion of the communications software which are given in the Appendix D - Communications Software Listing.

Conversely if one stop bit is chosen for five bits per character, it is converted to 1.5 stop bits.

2. SERIAL.ASM

```
        TITLE   SERIAL.ASM
SERIAL_VERSION  EQU     "V02Jun93.01"

;
; SERIAL.ASM    - serial port handlers
;               - timer support
;               - control-C and Break trapping
;               - critical error trapping to allow clean exit on Abort
;

;=============================================================================
;               'C' Language Interface
;-----------------------------------------------------------------------------

        PUBLIC  _open_ser,_close_ser,_stat_ser,_ver_ser,_set_ser
        PUBLIC  _read_ser,_write_ser

; Serial Port Routines
; --------------------
;   int open_ser(int port,int type)     Opens serial 'port',valid range is 1-10
;                                       for COM1-10.  'type' is 0 for standard
;                                       port addresses, 1 for Digiboard.
;                                       Returns 0=OK, 1=port out of range
;
;   int close_ser(int port)             Close serial 'port', valid range is 1-10
;                                       returns 0 for OK,1 for port out of range
;
;   int stat_ser(void)                  Returns composite status of ports.  See
;                                       the equates for status bit definitions
;
;   char *ver_ser(void)                 Returns string showing version number.
;
;   int set_ser(int port,int parm)      Sets baud rate, bits, stop and parity
;                                       See equate for definitions of bits in
;                                       'parm'.  Valid 'port' is 1-10.
;                                       Returns 0 = OK, 1 = port out of range
;
;   int read_ser(int port)              Get character from 'port', valid range
;                                       is 1-10.  Returns character, -1 for no
;                                       char avail or -2 for 'port' out of range
;
;   int write_ser(int port, int ich)    Sends 'ich' to 'port', valid range 1-10
;                                       Returns 0 = OK, 1 = port out of range
;-------------

        PUBLIC  _open_time,_close_time,_set_time,_chk_time

; Timer Support Routines
; ----------------------
;   void open_time(void)        Initializes and enables all countdown timers
```

```
;
;  void close_time(void)       Disables all countdown timers
;
;  int set_time(int timer, int tick)   Sets countdown 'timer' to value 'tick'
;                                        'Tick' units = 1/16.7s  Valid 'timer'
;                                        is 0-10.  Returns 0=OK, 1=out of range
;
;  int chk_time(int timer)     Returns value of countdown 'timer'.  Valid range
;                                0-10.  Returns 0 when countdown complete, -1 if
;                                'timer' out of range
;--------

        PUBLIC  _open_break,_close_break,_press_break

; Control-C and Break Handling Routines
; -------------------------------------
;   void open_break(void)      Initializes and enables Cntl-C/Break handler
;
;   void close_break(void)     Restores system Cntl-C/Break handler
;
;   int press_break(void)      Returns non-zero if Cntl-C/Break pressed since
;                                last call otherwise returns 0
;--------

        PUBLIC  _open_crit,_close_crit

; Critical Error Handling Routines
; --------------------------------
;   void open_crit(void)       Enables critical error handler.  This allows
;                                a clean exit if Abort is chosen
;
;
;   void close_crit(void)      Restores system critical error handler
;--------


; Memory Model Size
;------------------
        .MODEL  SMALL
Arg1    EQU     [BP+4]          ; [BP+6] for large model
Arg2    EQU     [BP+6]          ; [BP+8]


;===============================================================================
;               Common Declarations
;-------------------------------------------------------------------------------

MINCOM  EQU     COM1            ; Range of COMs to shut down if Abort is
MAXCOM  EQU     COMA            ;   chosen in critical error handler
NO      EQU     0               ; Interrupt initialized flag values
YES     EQU     1
NORMAL  EQU     0               ; Return values for routines
ERROR   EQU     1               ;  (not valid for read_ser and chk_timer)

;===============================================================================
;               Serial Port Handler
;-------------------------------------------------------------------------------

; Valid COM ports (see MINCOM and MAXCOM above)
COM1    EQU     1
COM2    EQU     2
COM3    EQU     3
COM4    EQU     4
COM5    EQU     5
COM6    EQU     6
COM7    EQU     7
COM8    EQU     8
```

```
COM9    EQU     9
COMA    EQU     10

; Definitions for the 8259 interrupt controller
OCW     EQU     20h                     ; Control word register
        EOI     EQU     20h             ; Nonspecific end-of-interrupt
IMR     EQU     21h                     ; Interrupt mask register

; Port offsets for UART registers
S_RXD   EQU     0                       ; Receive data register (R,DLAB=0)
S_TXD   EQU     0                       ; Transmit data register (W,DLAB=0)
S_DLSB  EQU     0                       ; Baud rate divisor LSB (W,DLAB=1)
S_DMSB  EQU     1                       ; Baud rate divisor MSB (W,DLAB=1)
S_IER   EQU     1                       ; Interrupt enable register (DLAB=0)
        DISINT  EQU     00000000b               ; Disable all interrupts
        ENRXD   EQU     00000001b               ; Enable Rx data interrupts
        ENTXD   EQU     00000010b               ; Enable Tx empty interrupts
        ENBRK   EQU     00000100b               ; Enable Break/Error ints
        ENCTL   EQU     00001000b               ; Enable Control line ints
S_IIR   EQU     2                       ; Interrupt identification register
        CTLINE  EQU     00000000b               ; Control line int
        NOINTS  EQU     00000001b               ; No interrupts occurred
        TXDRDY  EQU     00000010b               ; Tx empty interrupt
        RXDRDY  EQU     00000100b               ; Rx data interrupt
        BREAKE  EQU     00000110b               ; Break/Error interrupt
        VALBIT  EQU     00000111b               ; Valid bit mask
S_LCR   EQU     3                       ; Line control register
        BIT5    EQU     00000000b               ; Number of bits/character
        BIT6    EQU     00000001b
        BIT7    EQU     00000010b
        BIT8    EQU     00000011b
        STOP1   EQU     00000000b               ; Number of stop bits
        STOP2   EQU     00000100b
        PARNO   EQU     00000000b               ; Parity (none)
        PARODD  EQU     00001000b                       ; Odd
        PAREVN  EQU     00011000b                       ; Even
        PAR0    EQU     00111000b                       ; Force 0
        PAR1    EQU     00101000b                       ; Force 1
        BRKOFF  EQU     00000000b               ; Disable break
        BRKON   EQU     01000000b               ; Send break
        DLAB    EQU     10000000b               ; Controls divisor (addr 0/1)
S_MCR   EQU     4                       ; Modem control register
        DTR     EQU     00000001b               ; Set Data Terminal Ready
        RTS     EQU     00000010b               ; Set Request To Send
        OUT1    EQU     00000100b               ; Set out 1 (reset Hayes modem)
        OUT2    EQU     00001000b               ; Set out 2 (enable interrupts)
        LOOPBK  EQU     00010000b               ; Set loopback mode
S_LSR   EQU     5                       ; Line status register
        RXREDY  EQU     00000001b               ; Rx data character available
        OVERUN  EQU     00000010b               ; Overrun error
        PARITY  EQU     00000100b               ; Parity error
        FRAME   EQU     00001000b               ; Framing error
        BREAK   EQU     00010000b               ; Break received
        TXREDY  EQU     00100000b               ; Tx hold register empty
        TXSRDY  EQU     01000000b               ; Tx shift register empty
S_MSR   EQU     6                       ; Modem status register
        DELCTS  EQU     00000001b               ; Change in CTS line
        DELDSR  EQU     00000010b               ; Change is DSR line
        FALRI   EQU     00000100b               ; Falling edge of RI line
        DELCD   EQU     00001000b               ; Change in CD line
        CTS     EQU     00010000b               ; State of CTS line
        DSR     EQU     00100000b               ; State of DSR line
        RI      EQU     01000000b               ; State of RI line
        CD      EQU     10000000b               ; State of CD line

; Status bits for variable stat - returned by stat_ser()
```

```
INVINT  EQU     00000001b       ; Interrupt called, int bit not set      (1)
HANDSK  EQU     00000010b       ; Handshaking line change                (2)
BRKERR  EQU     00000100b       ; Error or break occured                 (4)
RXOVER  EQU     00001000b       ; Receive buffer overflow                (8)
TXOVER  EQU     00010000b       ; Transmit buffer overflow              (16)
TXFULL  EQU     00100000b       ; Transmit buffer not empty, valid only (32)
                                ;  on return from stat_ser()

; Rx and Tx buffer definistions
BSIZE   EQU     512             ; Buffer size
BFLOW   EQU     BSIZE-4         ; Overflow point for buffer


.DATA
s_base  DW      ?               ; Base address of serial port
stat    DW      0               ; Status
eoi_cnt DW      0               ; Number of EOIs in the isr

bas_tbl LABEL WORD              ; Serial port base addresses
        DW      03F8h                   ; COM1
        DW      02F8h                   ; COM2
bas_c3  DW      0100h                   ; COM3
bas_c4  DW      0108h                   ; COM4
        DW      0110h                   ; COM5
        DW      0118h                   ; COM6
        DW      0120h                   ; COM7
        DW      0128h                   ; COM8
        DW      0130h                   ; COM9
        DW      0138h                   ; COMA

t_tbl   LABEL WORD              ; Translation table:  serial port -> IRQ
        DW      2                       ; COM1, IRQ4
        DW      0                       ; COM2, IRQ3
t_c3    DW      0                       ; COM3, IRQ3
t_c4    DW      0                       ; COM4, IRQ3
        DW      0                       ; COM5, IRQ3
        DW      0                       ; COM6, IRQ3
        DW      0                       ; COM7, IRQ3
        DW      0                       ; COM8, IRQ3
        DW      0                       ; COM9, IRQ3
        DW      0                       ; COMA, IRQ3

use_tbl LABEL   WORD    ; Table for flags to indicate port in use
        DW      NO              ; COM1
        DW      NO              ; COM2
        DW      NO              ; COM3
        DW      NO              ; COM4
        DW      NO              ; COM5
        DW      NO              ; COM6
        DW      NO              ; COM7
        DW      NO              ; COM8
        DW      NO              ; COM9
        DW      NO              ; COMA

cnt_tbl LABEL   WORD    ; Table to count number of ports using IRQs
        DW      0               ; IRQ3
        DW      0               ; IRQ4

off_tbl LABEL   WORD    ; Table of offsets for old vectors
        DW      ?               ; IRQ3
        DW      ?               ; IRQ4

seg_tbl LABEL   WORD    ; Table of segments for old vectors
        DW      ?               ; IRQ3
        DW      ?               ; IRQ4
```

```
eoi_flg LABEL   WORD    ; Table of flags for eoi services
        DW      0               ; IRQ3
        DW      0               ; IRQ4

rx_put  LABEL   WORD            ; Receive buffer put pointers
        DW      rx1_buf                 ; COM1
        DW      rx2_buf                 ; COM2
        DW      rx3_buf                 ; COM3
        DW      rx4_buf                 ; COM4
        DW      rx5_buf                 ; COM5
        DW      rx6_buf                 ; COM6
        DW      rx7_buf                 ; COM7
        DW      rx8_buf                 ; COM8
        DW      rx9_buf                 ; COM9
        DW      rxa_buf                 ; COMA

rx_get  LABEL   WORD            ; Receive buffer get pointers
        DW      rx1_buf                 ; COM1
        DW      rx2_buf                 ; COM2
        DW      rx3_buf                 ; COM3
        DW      rx4_buf                 ; COM4
        DW      rx5_buf                 ; COM5
        DW      rx6_buf                 ; COM6
        DW      rx7_buf                 ; COM7
        DW      rx8_buf                 ; COM8
        DW      rx9_buf                 ; COM9
        DW      rxa_buf                 ; COMA

rx_cnt  LABEL   WORD            ; Receive buffer character counts
        DW      0                       ; COM1
        DW      0                       ; COM2
        DW      0                       ; COM3
        DW      0                       ; COM4
        DW      0                       ; COM5
        DW      0                       ; COM6
        DW      0                       ; COM7
        DW      0                       ; COM8
        DW      0                       ; COM9
        DW      0                       ; COMA

rx_beg  LABEL   WORD            ; Pointer to beginning of receive buffer
        DW      rx1_buf                 ; COM1
        DW      rx2_buf                 ; COM2
        DW      rx3_buf                 ; COM3
        DW      rx4_buf                 ; COM4
        DW      rx5_buf                 ; COM5
        DW      rx6_buf                 ; COM6
        DW      rx7_buf                 ; COM7
        DW      rx8_buf                 ; COM8
        DW      rx9_buf                 ; COM9
        DW      rxa_buf                 ; COMA

rx_end  LABEL   WORD            ; Pointer to end of receive buffer
        DW      rx1_lst                 ; COM1
        DW      rx2_lst                 ; COM2
        DW      rx3_lst                 ; COM3
        DW      rx4_lst                 ; COM4
        DW      rx5_lst                 ; COM5
        DW      rx6_lst                 ; COM6
        DW      rx7_lst                 ; COM7
        DW      rx8_lst                 ; COM8
        DW      rx9_lst                 ; COM9
        DW      rxa_lst                 ; COMA

tx_put  LABEL   WORD            ; Transmit buffer put pointers
        DW      tx1_buf                 ; COM1
```

```
        DW      tx2_buf                 ; COM2
        DW      tx3_buf                 ; COM3
        DW      tx4_buf                 ; COM4
        DW      tx5_buf                 ; COM5
        DW      tx6_buf                 ; COM6
        DW      tx7_buf                 ; COM7
        DW      tx8_buf                 ; COM8
        DW      tx9_buf                 ; COM9
        DW      txa_buf                 ; COMA

tx_get  LABEL   WORD            ; Transmit buffer get pointers
        DW      tx1_buf                 ; COM1
        DW      tx2_buf                 ; COM2
        DW      tx3_buf                 ; COM3
        DW      tx4_buf                 ; COM4
        DW      tx5_buf                 ; COM5
        DW      tx6_buf                 ; COM6
        DW      tx7_buf                 ; COM7
        DW      tx8_buf                 ; COM8
        DW      tx9_buf                 ; COM9
        DW      txa_buf                 ; COMA

tx_cnt  LABEL   WORD            ; Transmit buffer character counts
        DW      0                       ; COM1
        DW      0                       ; COM2
        DW      0                       ; COM3
        DW      0                       ; COM4
        DW      0                       ; COM5
        DW      0                       ; COM6
        DW      0                       ; COM7
        DW      0                       ; COM8
        DW      0                       ; COM9
        DW      0                       ; COMA

tx_beg  LABEL   WORD            ; Pointer to beginning of transmit buffer
        DW      tx1_buf                 ; COM1
        DW      tx2_buf                 ; COM2
        DW      tx3_buf                 ; COM3
        DW      tx4_buf                 ; COM4
        DW      tx5_buf                 ; COM5
        DW      tx6_buf                 ; COM6
        DW      tx7_buf                 ; COM7
        DW      tx8_buf                 ; COM8
        DW      tx9_buf                 ; COM9
        DW      txa_buf                 ; COMA

tx_end  LABEL   WORD            ; Pointer to end of transmit buffer
        DW      tx1_lst                 ; COM1
        DW      tx2_lst                 ; COM2
        DW      tx3_lst                 ; COM3
        DW      tx4_lst                 ; COM4
        DW      tx5_lst                 ; COM5
        DW      tx6_lst                 ; COM6
        DW      tx7_lst                 ; COM7
        DW      tx8_lst                 ; COM8
        DW      tx9_lst                 ; COM9
        DW      txa_lst                 ; COMA

; Receive buffers
rx1_buf DB      BSIZE DUP (?)           ; COM1
rx1_lst EQU     $
rx2_buf DB      BSIZE DUP (?)           ; COM2
rx2_lst EQU     $
rx3_buf DB      BSIZE DUP (?)           ; COM3
rx3_lst EQU     $
rx4_buf DB      BSIZE DUP (?)           ; COM4
```

```
rx4_lst EQU     $
rx5_buf DB      BSIZE DUP (?)           ; COM5
rx5_lst EQU     $
rx6_buf DB      BSIZE DUP (?)           ; COM6
rx6_lst EQU     $
rx7_buf DB      BSIZE DUP (?)           ; COM7
rx7_lst EQU     $
rx8_buf DB      BSIZE DUP (?)           ; COM8
rx8_lst EQU     $
rx9_buf DB      BSIZE DUP (?)           ; COM9
rx9_lst EQU     $
rxa_buf DB      BSIZE DUP (?)           ; COMA
rxa_lst EQU     $

; Transmit buffers
tx1_buf DB      BSIZE DUP (?)           ; COM1
tx1_lst EQU     $
tx2_buf DB      BSIZE DUP (?)           ; COM2
tx2_lst EQU     $
tx3_buf DB      BSIZE DUP (?)           ; COM3
tx3_lst EQU     $
tx4_buf DB      BSIZE DUP (?)           ; COM4
tx4_lst EQU     $
tx5_buf DB      BSIZE DUP (?)           ; COM5
tx5_lst EQU     $
tx6_buf DB      BSIZE DUP (?)           ; COM6
tx6_lst EQU     $
tx7_buf DB      BSIZE DUP (?)           ; COM7
tx7_lst EQU     $
tx8_buf DB      BSIZE DUP (?)           ; COM8
tx8_lst EQU     $
tx9_buf DB      BSIZE DUP (?)           ; COM9
tx9_lst EQU     $
txa_buf DB      BSIZE DUP (?)           ; COMA
txa_lst EQU     $

.CONST
ver_str DB      SERIAL_VERSION,0        ; Version number string
isr_vec DD      ser_int                 ; Pointer to ISR

baud_dv LABEL WORD              ; Baud rate divisor table
        DW      0417h                   ; 110 bps
        DW      0300h                   ; 150 bps
        DW      0180h                   ; 300 bps
        DW      00C0h                   ; 600 bps
        DW      0060h                   ; 1200 bps
        DW      0030h                   ; 2400 bps
        DW      0018h                   ; 4800 bps
        DW      000Ch                   ; 9600 bps

                                ; Standard COM port definitions
sbas_c3 DW      03E8h                   ; COM3
sbas_c4 DW      02E8h                   ; COM4

                                ; Standard COM IRQ definitions
st_c3   DW      2                       ; COM3, IRQ4
st_c4   DW      0                       ; COM4, IRQ3

get_tbl LABEL WORD              ; Serial port ISR "get vector" commands
        DW      350Bh                   ; IRQ3
        DW      350Ch                   ; IRQ4

put_tbl LABEL WORD              ; Serial port ISR "put vector" commands
        DW      250Bh                   ; IRQ3
        DW      250Ch                   ; IRQ4
```

```
dis_tbl LABEL WORD              ; 8259 masks to disable serial interrupts
        DW      00001000b               ; IRQ3
        DW      00010000b               ; IRQ4

en_tbl  LABEL WORD              ; 8259 masks to enable serial interrupts
        DW      11110111b               ; IRQ3
        DW      11101111b               ; IRQ4

eoi_tbl LABEL WORD              ; 8259 specific end-of-interrupts
        DW      63h                     ; IRQ3
        DW      64h                     ; IRQ4


.CODE
_open_ser PROC NEAR     ; Install serial port ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get port number passed by C
        MOV     AH,Arg2         ; Get configuration passed by C
        CMP     BX,MAXCOM       ; Check to see if it is within range
        JLE     os1
        JMP     oserr
os1:    CMP     BX,MINCOM
        JGE     os2
        JMP     oserr
os2:    CMP     AH,0            ; If standard type, reassign COM3/4 addresses
        JNE     os3             ;  and IRQs over the Digiboard ones
        MOV     AX,sbas_c3
        MOV     bas_c3,AX
        MOV     AX,sbas_c4
        MOV     bas_c4,AX
        MOV     AX,st_c3
        MOV     t_c3,AX
        MOV     AX,st_c4
        MOV     t_c4,AX
os3:    DEC     BX              ; Convert port number to table pointer
        SAL     BX,1
        CLI                     ; Disable interrupts while changing vectors
        MOV     AX,use_tbl[BX]  ; Check to see if it is already open
        CMP     AX,NO
        JE      os4
        JMP     oserr
os4:    MOV     use_tbl[BX],YES         ; Set used flag
        MOV     AX,bas_tbl[BX]          ; Set up serial port base address
        MOV     s_base,AX
        MOV     AL,ENRXD+ENTXD+ENBRK+ENCTL      ; Enable all interrupts
        MOV     DX,S_IER
        ADD     DX,s_base
        OUT     DX,AL
clrdat: MOV     DX,S_IIR        ; Clear junk from UART
        ADD     DX,s_base
        IN      AL,DX           ; Check for unserviced interrupts
        MOV     AH,AL
        TEST    AL,NOINTS
        JNZ     clrok
        CMP     AH,CTLINE       ; If control line interrupt pending
        JNE     os5             ;   then read MSR to clear it
        MOV     DX,S_MSR
        ADD     DX,s_base
        IN      AL,DX
os5:    CMP     AH,TXDRDY       ; If Tx empty interrupt pending
```

```
        JNE     os6             ;   then do nothing
os6:    CMP     AH,RXDRDY       ; If Rx data interrupt pending
        JNE     os7             ;   then read data
        MOV     DX,S_RXD
        ADD     DX,s_base
        IN      AL,DX
os7:    CMP     AH,BREAKE       ; If Break/Error interrupt pending
        JNE     clrdat          ;   then read LSR to clear it
        MOV     DX,S_LSR
        ADD     DX,s_base
        IN      AL,DX
        JMP     clrdat          ; Check for more pending interrupts
clrok:  MOV     AL,DTR+RTS+OUT2 ; Set all handshaking lines
        MOV     DX,S_MCR
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AL,ENRXD+ENTXD  ; Enable Rx and Tx interrupts
        MOV     DX,S_IER
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AX,t_tbl[BX]    ; Translate port number to IRQ number
        MOV     BX,AX
        MOV     AX,cnt_tbl[BX]  ; See if IRQ is already initialized
        INC     cnt_tbl[BX]
        CMP     AX,0
        JG      osok
        MOV     AX,get_tbl[BX]  ; Get old interrupt vector
        PUSH    BX
        INT     21h
        MOV     AX,BX
        POP     BX
        MOV     off_tbl[BX],AX  ; Save for restoring later
        MOV     seg_tbl[BX],ES
        PUSH    DS
        MOV     AX,put_tbl[BX]  ; Put in new int vector
        LDS     DX,isr_vec      ; DS:DX point to new ISR
        INT     21h
        POP     DS
        IN      AL,IMR                  ; Enable 8259 PIC
        AND     AL,BYTE PTR en_tbl[BX]
        OUT     IMR,AL
        MOV     AL,EOI                  ; Send out an EOI to clear it
        OUT     OCW,AL
osok:   MOV     AX,NORMAL       ; Normal return
        JMP     SHORT osdone
oserr:  MOV     AX,ERROR        ; Error return
osdone: STI                     ; Re-enable interrupts
        POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_open_ser ENDP


_close_ser PROC NEAR    ; Remove serial port ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get port number passed by C
        CMP     BX,MAXCOM       ; Ensure it is within range
        JLE     cs1
```

```
        JMP     cserr
cs1:    CMP     BX,MINCOM
        JGE     cs2
        JMP     cserr
cs2:    DEC     BX              ; Convert port number to a table pointer
        SAL     BX,1
        MOV     AX,use_tbl[BX]  ; Get old value for used flag
        MOV     use_tbl[BX],NO  ; Clear used flag
        CMP     AX,YES          ; See if it was used before
        JE      cs3
        JMP     cserr           ; Port wasn't opened
cs3:    MOV     AX,bas_tbl[BX]  ; Get UART base address for port
        MOV     s_base,AX
        MOV     AX,t_tbl[BX]    ; Translate port number to IRQ number
        MOV     BX,AX
        DEC     cnt_tbl[BX]     ; Decrease count of ports using this IRQ
        JNZ     csok            ; If non-zero, do not disable IRQ
        IN      AL,IMR                  ; Disable COM interrupts in 8259
        OR      AL,BYTE PTR dis_tbl[BX]
        OUT     IMR,AL
        MOV     AL,DISINT               ; Disable UART interrupts
        MOV     DX,S_IER
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AX,put_tbl[BX]  ; Restore original vector
        MOV     DX,off_tbl[BX]
        MOV     CX,seg_tbl[BX]
        MOV     DS,CX
        INT     21h
csok:   MOV     AX,NORMAL       ; Normal return
        JMP     SHORT csdone
cserr:  MOV     AX,ERROR        ; Error return
csdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_close_ser ENDP


_stat_ser PROC NEAR     ; Get serial port and buffer status
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     AX,stat
        OR      AX,TXFULL       ; Set transmitter buffers full flag
        MOV     BX,MAXCOM       ; Convert max port # to table offset
        DEC     BX
        SAL     BX,1
sa1:    CMP     tx_cnt[BX],0    ; Check to see if any tx buffer has data
        JNE     sa2
        SUB     BX,2
        JGE     sa1
        XOR     AX,TXFULL       ; Reset tx buffers full flag
sa2:    AND     stat,00H        ; Clear status for next call
        POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_stat_ser endp
```

```
_ver_ser PROC NEAR      ; Returns string showing version number
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     AX,OFFSET ver_str
        POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_ver_ser endp


_set_ser PROC NEAR      ; Set serial port paramenters
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get port number passed by C
        MOV     AH,Arg2         ; Get configuration passed by C
        CMP     BX,MAXCOM       ; Ensure port in range
        JLE     ss1
        JMP     sserr
ss1:    CMP     BX,MINCOM
        JGE     ss2
        JMP     sserr
ss2:    DEC     BX              ; Convert port number to table pointer
        SAL     BX,1
        MOV     CX,bas_tbl[BX]  ; Get base address of UART
        MOV     s_base,CX
        MOV     AL,DLAB         ; Set DLAB bit to access divider regs
        MOV     DX,S_LCR
        ADD     DX,s_base
        OUT     DX,AL
        MOV     DL,AH           ; Shift configuration to BAUD field
        MOV     CL,4
        ROL     DL,CL
        AND     DX,00001110b            ; Mask out all other bits
        MOV     DI,OFFSET baud_dv
        ADD     DI,DX                   ; Convert to table pointer
        MOV     AL,[DI+1]       ; Set high byte of divider
        MOV     DX,S_DMSB
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AL,[DI]         ; Set low byte of divider
        MOV     DX,S_DLSB
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AL,AH           ; Use rest of configuration to set LCR
        AND     AL,00011111b
        MOV     DX,S_LCR
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AL,ENRXD+ENTXD  ; Enable Rx or Tx interrupts
        MOV     DX,S_IER
        ADD     DX,s_base
        OUT     DX,AL
        MOV     AX,NORMAL       ; Normal return
        JMP     SHORT ssdone
```

```
sserr:  MOV     AX,ERROR        ; Error return
ssdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_set_ser ENDP


_read_ser PROC NEAR     ; reads byte from serial port receive buffer
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get port number passed by C
        CMP     BX,MAXCOM
        JLE     rs1
        JMP     rserr
rs1:    CMP     BX,MINCOM       ; Ensure port is within range
        JGE     rs2
        JMP     rserr
rs2:    DEC     BX              ; Convert port to table pointer
        SAL     BX,1
        MOV     DI,rx_get[BX]   ; See if character is available
        CMP     DI,rx_put[BX]
        JE      nodata
        INC     DI              ; Advance (with wraparound) get pointer DI
        CMP     DI,rx_end[BX]
        JNE     rs3
        MOV     DI,rx_beg[BX]
rs3:    MOV     AL,[DI]         ; Get the character and clear upper byte
        MOV     AH,0
        MOV     rx_get[BX],DI   ; Save new get pointer
        DEC     rx_cnt[BX]      ; Reduce the buffer character count
        JMP     SHORT rsdone
rserr:  MOV     AX,-2           ; Error return - port number out of range
        JMP     SHORT rsdone
nodata: MOV     AX,-1           ; Error return - no data available
rsdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_read_ser ENDP


_write_ser PROC NEAR    ; Write char to serial port or tx buffer
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get port number passed by C
        CMP     BX,MAXCOM       ; Ensure port within range
        JLE     ws1
        JMP     wserr
ws1:    CMP     BX,MINCOM
        JGE     ws2
        JMP     wserr
ws2:    DEC     BX              ; Convert port to table pointer
        SAL     BX,1
```

```
        MOV     AX,bas_tbl[BX]  ; Get base address of UART
        MOV     s_base,AX
        MOV     DI,tx_put[BX]   ; See if buffer already has characters
        CMP     DI,tx_get[BX]
        JNE     sv_chr
;       MOV     DX,S_MSR        ; Check for DSR, CTS
;       ADD     DX,s_base
;       IN      AL,DX
;       AND     AL,CTS+DSR
;       CMP     AL,CTS+DSR
;       JNE     sv_chr
        MOV     DX,S_LSR        ; Check for UART ready
        ADD     DX,s_base
        IN      AL,DX
        TEST    AL,TXREDY
        JZ      sv_chr
        MOV     AL,Arg2         ; Transmit char from 'C'
        MOV     DX,S_TXD
        ADD     DX,s_base
        OUT     DX,AL
        jmp     SHORT wsok
sv_chr: MOV     AL,Arg2         ; Save character passed from C in buffer
        MOV     [DI],AL
        INC     DI              ; Advance (with wraparound) put pointer DI
        CMP     DI,tx_end[BX]
        JNE     ws3
        MOV     DI,tx_beg[BX]
ws3:    MOV     tx_put[BX],DI
        INC     tx_cnt[BX]              ; Check for transmit buffer overflow
        CMP     tx_cnt[BX],BFLOW
        JLE     wsok
        OR      stat,TXOVER             ; Set status bit for overflow
wsok:   MOV     AX,NORMAL       ; Normal return
        JMP     SHORT wsdone
wserr:  MOV     AX,ERROR        ; Error return
wsdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_write_ser ENDP


ser_int:                ; Serial port ISR for COM1-COM8 (IRQ3 & IRQ4)
        CLI
        PUSH    DS
        PUSH    ES
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    SI
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX

        MOV     BX,0            ; Start table pointer at first device
        MOV     eoi_cnt,BX      ; Clear counter and flags for IRQ3,4
        MOV     eoi_flg,BX
        MOV     eoi_flg+2,BX
chkdev: CMP     use_tbl[BX],NO  ; Check to see if in use
        JE      nxtdev
        MOV     SI,bas_tbl[BX]  ; Check to see if this UART caused int
        MOV     DX,S_IIR
        ADD     DX,SI
```

```
        IN      AL,DX
        AND     AX,VALBIT
        TEST    AX,NOINTS       ; If interrupt found then process
        JZ      found
nxtdev: ADD     BX,2            ; Next UART
        CMP     BX,MAXCOM*2
        JL      chkdev
isdone: CMP     eoi_cnt,0
        JE      notint
        JMP     sidone
notint: OR      stat,INVINT     ; No in-use UARTs caused interrupt,
        JMP     SHORT sidon1    ;  so set invalid interrupt status bit
found:  MOV     DI,AX           ; Use interrupt ID number as pointer
        JMP     CS:i_tbl[DI]
i_tbl   LABEL   WORD
        DW      ctlint
        DW      txint
        DW      rxint
        DW      brkint

ctlint: OR      stat,HANDSK     ; Handshaking line changed (set status bit)
        MOV     DX,S_MSR                ; Clear interrupt
        ADD     DX,SI
        IN      AL,DX
        JMP     SHORT sidon1

txint:  MOV     DI,tx_get[BX]   ; Tx empty
        CMP     DI,tx_put[BX]           ; If data in buffer
        JE      txend
        DEC     tx_cnt[BX]              ;   then decrement count
        MOV     AL,[DI]                 ;   and send it out
        MOV     DX,S_TXD
        ADD     DX,SI
        OUT     DX,AL
        INC     DI                      ; Advance get pointer (with wraparound)
        CMP     DI,tx_end[BX]
        JNE     txend
        MOV     DI,tx_beg[BX]
txend:  MOV     tx_get[BX],DI
        JMP     SHORT sidon1

rxint:  MOV     DX,S_RXD        ; Rx data available
        ADD     DX,SI                   ; Get character from UART
        IN      AL,DX
        MOV     DI,rx_put[BX]           ; Advance put pointer (with wraparound)
        INC     DI
        CMP     DI,rx_end[BX]
        JNE     ri1
        MOV     DI,rx_beg[BX]
ri1:    MOV     [DI],AL                 ; Put character in buffer
        MOV     rx_put[BX],DI
        INC     rx_cnt[BX]              ; Increment buffer count
        CMP     rx_cnt[BX],BFLOW        ; Check for receive buffer overflow
        JLE     rxend
        OR      stat,RXOVER             ; Set status bit
rxend:  JMP     SHORT sidon1

brkint: OR      stat,BRKERR     ; Break or error occurred (set status bit)
        MOV     DX,S_LSR                ; Clear interrupt
        ADD     DX,SI
        IN      AL,DX
        JMP     SHORT sidon1

sidon1: PUSH    BX
        MOV     AX,t_tbl[BX]            ; Translate port number to IRQ number
        MOV     BX,AX
```

```
        INC     eoi_cnt                 ; Count number of total eoi
        INC     eoi_flg[BX]             ; Set flag for later EOI
        POP     BX
        JMP     chkdev

sidone: CMP     eoi_flg,0       ; Check for EOI for first IRQ
        JE      si1
        MOV     AX,eoi_tbl              ; Get IRQ eoi instruction
        OUT     OCW,AL                  ; Send EOI to 8259 chip
si1:    CMP     eoi_flg+2,0     ; Check for EOI for second IRQ
        JE      si2
        MOV     AX,eoi_tbl+2            ; Get IRQ eoi instruction
        OUT     OCW,AL                  ; Send EOI to 8259 chip
si2:
        POP     SI
        POP     DI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        POP     DS
        IRET


;===========================================================================
;               Timer support
;---------------------------------------------------------------------------

NTIMER  EQU     11              ; Number of countdown timers
GETTIV  EQU     351Ch           ; Get timer interrupt vector
PUTTIV  EQU     251Ch           ; Put timer interrupt vector

.DATA
t_vec   DD      ?               ; Storage for original INT 1CH vector
t_init  DW      NO              ; Flag to indicate if initialized
count   LABEL   WORD            ; Table of count down values
        DW      NTIMER DUP (0)

.CODE
_open_time PROC NEAR    ; Install timer tick ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     t_init,NO       ; Check to see if already initialized
        JNE     SHORT otdone
        MOV     t_init,YES      ; Set initialized flag
        MOV     AX,GETTIV       ; Get interrupt vector for 1CH
        INT     21h
        MOV     WORD PTR t_vec,BX       ; Save old vector
        MOV     WORD PTR t_vec+2,ES
        MOV     AX,SEG time_int         ; DS:DX points to new routine
        MOV     DS,AX
        MOV     DX,OFFSET time_int
        MOV     AX,PUTTIV               ; Set interrupt vector
        INT     21h
otdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_open_time ENDP
```

```
_close_time PROC NEAR   ; Remove timer tick ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     t_init,YES      ; Check to see if initialized
        JNE     ctdone
        MOV     t_init,NO
        LDS     DX,t_vec        ; DS:DX points to original routine
        MOV     AX,PUTTIV       ; Set interrupt vector
        INT     21h
ctdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_close_time ENDP


_set_time PROC NEAR     ; Set the count-down timer counter
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get timer number passed by C
        CMP     BX,NTIMER       ; Ensure it is within range
        JL      st1
        JMP     sterr
st1:    CMP     BX,0
        JGE     st2
        JMP     sterr
st2:    SAL     BX,1            ; Convert timer number to table pointer
        MOV     AX,Arg2         ; Get the tick count passed by C
        MOV     count[BX],AX    ; Set countdown timer value
        MOV     AX,NORMAL
        JMP     SHORT stdone
sterr:  MOV     AX,ERROR        ; Error return
stdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_set_time ENDP


_chk_time PROC NEAR     ; Returns count-down value
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,Arg1         ; Get timer number passed by C
        CMP     BX,NTIMER       ; Ensure timer number is within range
        JL      ck1
        JMP     ckerr
ck1:    CMP     BX,0
        JGE     ck2
```

```
        JMP     ckerr
ck2:    SAL     BX,1            ; Convert timer number to table pointer
        MOV     AX,count[BX]    ; Load countdown value (0 if finished)
        JMP     SHORT ckdone
ckerr:  MOV     AX,-1           ; Error return - timer number out of range
ckdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_chk_time ENDP


time_int:               ; Timer tick interrupt service routine
        CLI
        PUSH    DS
        PUSH    ES
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     BX,0            ; Load table pointer for first timer
ti1:    DEC     count[BX]               ; Decrease count but not below 0
        JG      ti2
        AND     count[BX],0000h
ti2:    ADD     BX,2                    ; Get table pointer for next timer
        CMP     BX,NTIMER*2     ; Until done
        JL      ti1
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        POP     DS
        IRET


;==========================================================================
;               Control-C and Break Detection
;--------------------------------------------------------------------------

GETBIV  EQU     351Bh           ; Get Break interrupt vector
PUTBIV  EQU     251Bh           ; Put Break interrupt vector
GETCIV  EQU     3523h           ; Get Control-C interrupt vector
PUTCIV  EQU     2523h           ; Put Control-C interrupt vector

.DATA
b_vec   DD      ?               ; Storage for original INT 1BH vector
b_init  DW      NO              ; Flag to inidicated initialized
brkflg  DW      0               ; Flag that BREAK occurred

.CODE
_open_break PROC NEAR   ; Install control-C and break ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     b_init,NO       ; Check to see if initialized
        JNE     obdone
        MOV     b_init,YES      ; Set flag to indicate initialized
```

```
        MOV     AX,GETBIV               ; Get break interrupt vector
        INT     21h                     ;   (don't need to save for Control-C)
        MOV     WORD PTR b_vec,BX       ; Save break interrupt vector
        MOV     WORD PTR b_vec+2,ES
        MOV     AX,SEG break_int        ; DS:DX points to new break routine
        MOV     DS,AX
        MOV     DX,OFFSET break_int
        MOV     AX,PUTBIV               ; Set Break interrupt vector
        INT     21h
        MOV     AX,SEG ctlc_int         ; DS:DX points to new Control-C routine
        MOV     DS,AX
        MOV     DX,OFFSET ctlc_int
        MOV     AX,PUTCIV               ; Set Control-C interrupt vector
        INT     21h
obdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_open_break ENDP


_close_break PROC NEAR  ; Remove control-C and break ISR
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     b_init,YES      ; Check to see if initialized
        JNE     cbdone
        MOV     b_init,NO       ; Reset initialized flag
        LDS     DX,b_vec        ; DS:DX points to original
        MOV     AX,PUTBIV       ; Set Break interrupt vector
        INT     21h             ;  (system resets Control-C interrupt vector)
cbdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_close_break ENDP


_press_break PROC NEAR  ; Returns 0 if no break
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        XOR     AX,AX           ; Prepare to reset flag
        XCHG    AX,brkflg       ; Normal return         0000h = no break
        POP     DS                      ;               001Bh = Break
        POP     ES                      ;               0023h = Control-C
        MOV     SP,BP
        POP     BP
        RET
_press_break ENDP


break_int:              ; Control-break interrupt service routine
        PUSH    ES
        PUSH    DS
        PUSH    AX
```

```
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     brkflg,1Bh      ; Make it nonzero
        POP     AX
        POP     DS
        POP     ES
        IRET


ctlc_int:               ; Control-C interrupt service routine
        PUSH    ES
        PUSH    DS
        PUSH    AX
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        MOV     brkflg,23h      ; Make it nonzero
        POP     AX
        POP     DS
        POP     ES
        IRET


;===========================================================================
;               Critical Error Trapping
;---------------------------------------------------------------------------

GETEIV  EQU     3524h           ; Get critical error handler vector
PUTEIV  EQU     2524h           ; Put critical error handler vector

.DATA
e_vec   DD      ?               ; previous contents of crit error handler
e_init  DW      NO              ; Flag to indicate if initialized

.CONST
prompt  DB      0Dh,0Ah,'Critical Error Occurred: ',0Dh,0Ah
        DB      ' Abort, Retry, Ignore, Fail? ','$'

.CODE
_open_crit PROC NEAR    ; Install new critical error handler
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     e_init,NO       ; Check to see if initialized
        JNE     SHORT ocdone
        MOV     e_init,YES      ; Set initialized flag
        MOV     AX,GETEIV               ; Get old vector
        INT     21h
        MOV     WORD PTR e_vec,bx       ; Save old vector
        MOV     WORD PTR e_vec+2,es
        MOV     AX,SEG crit_hand        ; Set DS:DX to point to new handler
        MOV     DS,AX
        MOV     DX,OFFSET crit_hand
        MOV     AX,PUTEIV               ; Set up new handler
        INT     21h
ocdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_open_crit ENDP
```

```
_close_crit PROC NEAR   ; Restore original critical error handler
        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    DS
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
        CMP     e_init,YES      ; Check to see if initialized
        JNE     ccdone
        MOV     e_init,NO       ; Reset initialized flag
        LDS     DX,e_vec        ; Restor old vector
        MOV     AX,PUTEIV
        INT     21h
ccdone: POP     DS
        POP     ES
        MOV     SP,BP
        POP     BP
        RET
_close_crit ENDP

;
; This is the replacement critical error handler.  It
; prompts the user for Abort, Retry, Ignore, or Fail and
; returns the appropriate code to the MS-DOS kernel.
;
crit_hand PROC FAR      ; Critical error handler, called only by MS-DOS kernel
        PUSH    ES
        PUSH    DS
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
        MOV     AX,DGROUP
        MOV     DS,AX
        MOV     ES,AX
getkey: MOV     DX,OFFSET prompt        ; Display prompt for user
        MOV     AH,09h
        INT     21h
        MOV     AH,01h                  ; Get user's response
        INT     21h
        CMP     AL,'a'
        JE      dabort
        CMP     AL,'A'
        JE      dabort
        CMP     AL,'r'
        JE      dretry
        CMP     AL,'R'
        JE      dretry
        CMP     AL,'i'
        JE      dignor
        CMP     AL,'I'
        JE      dignor
        CMP     AL,'f'
        JE      dfail
        CMP     AL,'F'
        JE      dfail
        JMP     getkey
dabort: MOV     AL,2            ; Abort chosen
        CALL    _close_break            ; Restore Break/Control-C vector
        CALL    _close_time             ; Restore timer vector
        MOV     BX,MINCOM               ; Restore all serial vectors
```

```
d1:     CALL    _close_ser
        INC     BX
        CMP     BX,MAXCOM
        JLE     d1
        MOV     AL,2                    ; Set Abort return value
        JMP     ddone
dretry: MOV     AL,1            ; Retry chosen
        JMP     ddone
dignor: MOV     AL,0            ; Ignore chosen
        JMP     ddone
dfail:  MOV     AL,3            ; Fail chosen
        JMP     ddone
ddone:  POP     BP
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        POP     DS
        POP     ES
        IRET                    ; exit critical error handler
crit_hand ENDP

        END
```

References

[1] W. Stallings, *Data and Computer Communications*. New York, NY: Macmillan, 1985.

[2] R. Duncun, *The MS-DOS Encyclopedia*. Redmond, Washington: Microsoft Press, 1988.

[3] *Software Installation and Operation Manual for DigiCHANNEL PC/X, DigiCHANNEL MODEM/X and UNIX System V/386 Rel 3.2*. Eden Prairie, MN: DigiBoard, 1991.

[4] *Microsystem Components Handbook, Microprocessors Volume 1*. Santa Clara, CA: Intel Corporation, 1986.

[5] "INS8250, INS8250-B Universal Asynchronous Receiver/Transmitter Data Sheet", National Semiconductor.

UNCLASSIFIED

SECURITY CLASSIFICATION OF FORM
(highest classification of Title, Abstract, Keywords)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence Research Establishment Ottawa Ottawa, Ontario K1A 0Z4	**2. SECURITY CLASSIFICATION** (overall security classification of the document including special warning terms if applicable) UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C or U) in parentheses after the title.)

Real-time Interprocessor Serial Communications Software for Skynet EHF Trials (U)

4. AUTHORS (Last name, first name, middle initial)

Addison, Robin D.

5. DATE OF PUBLICATION (month and year of publication of document) July 1994	**6a. NO. OF PAGES** (total containing information. Include Annexes, Appendices, etc.) 145	**6b. NO. OF REFS** (total cited in document) 5

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.

DREO Report

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address.

Defence Research Establishment Ottawa
Ottawa, Ontario, K1A 0Z4

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant) 041LM and Project D6470	**9b. CONTRACT NO.** (if appropriate, the applicable number under which the document was written)
10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document.) DREO REPORT 1227	**10b. OTHER DOCUMENT NOS.** (Any other numbers which may be assigned this document either by the originator or by the sponsor)

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). however, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

Unlimited Announcement

13. **ABSTRACT** (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

The EHF (Extremely High Frequency) Skynet Trials consisted of several week-long accesses over Skynet 4A during 1993. The whole link (from transmitting ground terminal to Skynet to receiving ground terminal) was used to simulate an EHF downlink from a payload to a ground terminal. Use of the Skynet satellite allowed the experimentation at EHF with the ground terminal and payload simulators over a link that had real satellite effects such as link degradations caused by satellite motion and weather. To conduct the trials, it was recognized that many tasks needed to be active at once: pointing of antennas, monitoring power levels, synchronization, data communications and result logging. To shorten development time and simplify integration requirements, a distributed processing system (multiple computers) was chosen.

This paper describes the communications software which provided the services necessary for the distributed processing used in the trials. The challenge was to develop a system that was easy to integrate with the user software as well as to ensure that the communications hardware and software did not conflict with special purpose boards in the various computers. For simplicity, stop-and-wait ARQ (Automatic Repeat Request) protocol was used for high-level message passing. Low-level communications services that do not require handshaking, were also provided for equipment control. The communications software package met these challenges and after extensive testing, was proven to provide the necessary communications among all the processors of the distributed system.

14. **KEYWORDS, DESCRIPTORS or IDENTIFIERS** (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

serial communications
real-time
EHF
ARQ